xyz.py

someonesdad1@gmail.com 3 Mar 2013

The `xyz.py` script is a python script that provides a mini-language to perform analytical geometry calculations with points, lines, and planes in two-dimensional and three-dimensional Euclidean space.

It's unnecessary to know much analytical geometry to be able to use this script. If you're comfortable with the notion of locating objects in the plane or in space with Cartesian coordinates, you'll be able to do useful things with the script. I suggest going to the **Examples** section to get a feel for the type of problems that can be solved.

Table of Contents

Introduction

The `xyz.py` script lets you define points, lines, and planes in Euclidean space and display their coordinates and relationships in different coordinate systems. That sounds pretty dry and not terribly useful, so it's worthwhile to see some examples. I recommend that you first peruse the **Basic capabilities** example and read through the following features list. These two things should give you a high-level view of the capabilities of the script. Then when you're examining other reading material, refer to the **Commands** for details on the different commands.

Here's a brief history of this script. I first wrote a script I called `xy.py` to perform these types of calculations for two-dimensional analytical geometry problems. My canonical use case was to deal with survey information that I had taken from my yard. Naturally, it then occurred to me that it would be nice to have the script deal with three-dimensional information, as I could see occasional needs for such things (for example, I've used such things in the shop for drilling holes at compound angles; see the section **Shop problem** for an example. The three-dimensional script didn't share much of the code from the two-dimensional script (the area of a polygon and a circle from three points were the two main things I ported over), but as is common in developing software, the process of designing and writing the first version helped quite a bit in the second version.

One thing that surprised me about this script is that I probably spent more time designing the commands and their operation than I did on the implementation. It's easy to throw a bunch of commands together, but it's hard to make them into a useful, unified whole. There are currently around 50 different commands -- too many, in my opinion. However, I don't see an easy way of reducing that number significantly. Time will tell if I did a decent job or not.

Features

1. **Input from stdin or file**: input is taken from a file (the datafile) or stdin. You can add comments to the file to explain the details of the input information. Then the datafile can be saved in a revision control system to e.g. document how the coordinates of something were calculated.
2. **2D and 3D problems**: it lets you work in a familiar two-dimensional or three-dimensional environment. For example, you can work plane analytical geometry problems and see two-dimensional output (i.e., you won't see references to the third spatial coordinate z unless an object's z coordinate is nonzero).
3. **Transformations**: it supports the following transformations of the coordinate system: translations, rotations, and dilatations (i.e., scaling transformations). These transformations allow you to enter data in a convenient coordinate system, then use a different coordinate system for output. The coordinate systems can be pushed and popped from an internal stack, making it easy to use temporary coordinate systems without messing up ones already defined.
4. **Angular conveniences**: a variety of angular measurement modes and conventions are provided to make the statement and output of problems more familiar.
5. **Familiar coordinate systems**: you can print the output in Cartesian (rectangular), cylindrical, or spherical coordinates.
6. **Assign variables**: you can define variables for use later in the datafile.
7. **Expressions**: input data can be arbitrary python expressions.
8. **In-line python code**: you can intersperse your coordinate data and commands with arbitrary python code. All the variables and objects you define are available to the arbitrary python code.
9. **Uncertainty propagation**: you can include uncertainties in coordinate positions and see these uncertainties propagated into the results (see [gum] for more details). This requires the python uncertainties library (its use is optional).

10. **Familiar geometric objects**: points, lines, and planes are the geometrical objects supported (more may be added later).
11. **Vectors**: points, lines, and planes can be interpreted in a natural way as (bound) vectors, allowing you to use vector algebra in expressions.
12. **Geometric relationships**: you can find the intersections between the geometrical objects, distances between them, and angles between lines and planes.
13. **Projection and reflection**: points and lines can be projected into an arbitrary plane or reflected about the origin, a line (in the xy plane only), or a plane. Both orthogonal and perspective projections are supported.
14. **Attributes**: objects are provided with attributes that allow you to change the coordinates of a geometrical object in the current coordinate system.

Other possible features for the script have crossed my mind. One would be to add support for more geometrical objects (circles, disks, spheres, ellipses, etc.). Another would be to allow velocity and acceleration information to be given to the geometrical objects and to allow position calculations to take place at a desired time.

Notation

I'll use the usual vector notation for most things; vectors will be in a bold font. Unless otherwise stated, $\boldsymbol{a}$ represents a vector and a represents its magnitude. $\boldsymbol{a}\cdot\boldsymbol{b}$ denotes a scalar (dot) product and $\boldsymbol{a}\times\boldsymbol{b}$ denotes a vector (cross) product.

All coordinate systems are right-handed by default. You can change to a left-handed coordinate system if you wish by using an odd inversion with the **scale** command.

Glossary and symbols (in particular, note that the domain of coordinate angles may be different than the customary mathematical definitions):

$\hat{\boldsymbol{n}}$	Unit vector
$(-\pi/2,\pi/2]$	Half-open interval. Parentheses denote an open interval (i.e., the endpoint of the interval is not included in the interval); square brackets denote the interval is closed.
$\boldsymbol{r}$	Position vector
$\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$	Cartesian unit vectors (note they're used without a caret).
$\mathbb{R}_x$	Rotation matrix (the subscript indicates a rotation around the x axis)
CCS	Current coordinate system. The CCS is defined as a single transformation of the default coordinate system; this transformation is the result of all the `rotate`, `translate`, and `scale` commands that have been encountered up to this point.
Compass mode	The angle θ in cylindrical or spherical coordinates is measured clockwise from the +y axis when looking down on the origin from the +z direction (conventionally called "north"; east will be 90°). This is the same as the azimuths displayed by compasses that read from 0° to 360°.
CTM	A 4x4 coordinate transformation matrix. Its general form in the script is $\begin{bmatrix} a & b & c & t \\ d & e & f & u \\ g & h & i & v \\ 0 & 0 & 0 & 1 \end{bmatrix}$ where *a*-*i* and *t*, *u*, *v* are real constants. It left-multiplies homogeneous coordinates that are column vectors.
DCS	Default coordinate system.

Default coordinate system	The coordinate system that is implicitly defined when the `xyz.py` script is started.
Elevation mode	The spherical coordinate ϕ is replaced by ψ, which is an elevation angle above or below the xy plane (it is negative below the xy plane). Turning on compass mode and elevation mode simultaneously simulates altazimuth mode used in astronomy (the elevation is also called altitude).
homogeneous coordinates	To use matrices to represent affine transformations, it is convenient to use homogeneous coordinates. These are column vectors such as $[x\ y\ z\ 1]^T$.
Negative mode	When on, the θ angle in cylindrical or polar coordinates is measured in the opposite direction to normal. Thus, for regular polar coordinates, angles are increasing clockwise and for compass mode, angles are increasing counterclockwise (in all cases, the stated rotation sense is seen if the unit vector specifying the rotation axis has its tip hit you in the eye).
phi, ϕ	Spherical coordinate that is the angle from the +z axis. Will be on the interval $[0, \pi]$. Note this is different than the usual definition, which is usually the half-open interval $[0, \pi)$. For example, the point with Cartesian coordinates (0, 0, -1) has spherical coordinates `<<1, 0, 180 D>>` in degree measure (this is because `atan2(0, -1)` is π radians).
psi, ψ	Spherical coordinate that is the complement of ϕ (i.e., $\pi/2-\phi$); it measures the elevation above or below the xy plane. Will be on the interval $[-\pi/2, \pi/2]$.
r	Spherical coordinate radial dimension.
rho, ρ	Cylindrical coordinate radial distance in xy plane.
theta, θ	Spherical and cylindrical azimuthal coordinate in the xy plane. Will have the domain $[0, 2\pi)$.
x, y, z	Cartesian rectangular coordinates
xy, xz, yz	Cartesian coordinate planes containing the origin and perpendicular to the axis of the missing coordinate.

Angle measure

When Point objects are printed, you'll see their values in the current coordinate system and the type of coordinates you've chosen, either rectangular, cylindrical, or spherical. For the default angle measure of degrees, you'll see the point (1, 2, 3) given as follows:

Rectangular (x, y, z):	`Pt(1, 2, 3)`
Cylindrical (ρ, θ, z):	`Pt<2.24, 63.4, 3 o>`
Spherical (r, θ, ϕ):	`Pt<<3.74, 63.4, 36.7 o>>`

Note the cylindrical and spherical coordinates are surrounded by brackets that are the same as used for providing input in those coordinates; this notation is intended to alert you to the coordinate system being used. The `o` indicates degrees.

Different conventions are used for convenience in angular measurement; these are set by various commands such as **compass**, **deg**, **elev**, **neg**, **rad**, and **rev**.

The algorithms for azimuthal angle adjustment are: To convert θ from normal polar angle measurement (on the half-interval of [0, 2π)) to compass measurement (`compass on`), subtract θ from $\pi/2$ and add 2π if the number is negative. To convert θ from normal polar angle measurement to **neg** mode, take the negative of the angle and add 2π if the number is negative.

If elevation mode is on (`elev on`), then the canonical spherical coordinate ϕ is converted to the

elevation coordinate ψ by $\psi = \pi/2 - \phi$.

The following characters are used to denote settings in printed output:

- o Angle measure is in degrees. There is no corresponding letter for radians.
- V Angle measure is in revolutions.
- S Angle measure is special (i.e., not degrees, radians, or revolutions).
- C Compass mode is on (the polar angle θ is measured from the +y axis and increases clockwise).
- E Elevation mode is on (ψ is displayed for spherical coordinates instead of ϕ).
- \- Negative mode is on (angles' increasing sense is opposite to conventional).

If you see a θ value printed out as e.g. 360.0°, you'll know that it's a number just slightly smaller than 360 that rounds to 360 at the number of significant figures being used.

Important: if you use compass, elevation, or negative mode for output, the same angle conventions are used for the input commands and getting/setting the angular attributes of objects.

Examples

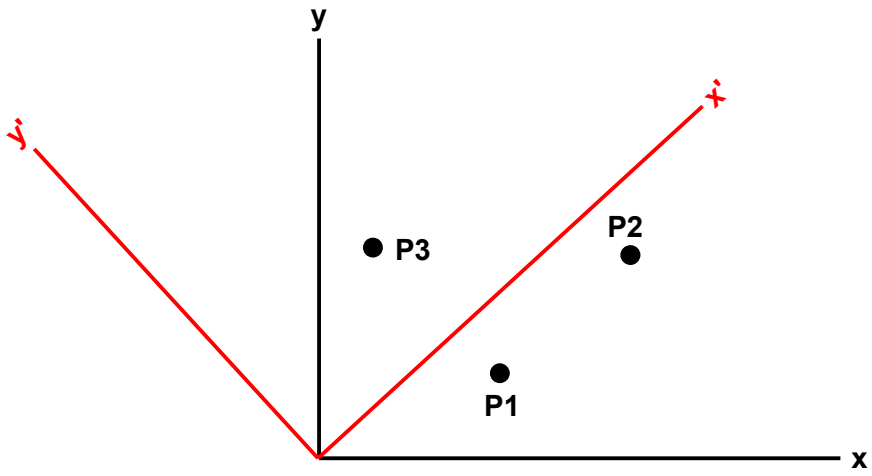
Two-dimensional examples

Basic capabilities

Features: entering points and printing them, coordinate transformations, changing units of measure using the `scale` command.

This example illustrates a basic problem that can be solved with the `xyz.py` script: **entering objects' coordinates in one coordinate system, then transforming the coordinate system and getting the objects' coordinates in a new coordinate system**.

Suppose I have three points `P1`, `P2`, and `P3` and I want their coordinates in a coordinate system that is rotated 40° from the first coordinate system (i.e., the red system), as in the following picture (the drawing is a sketch and is not to scale)



Suppose `P1` is (1.2, 0.6), `P2` = (2, 1.2), and `P3` = (3/4, 1.3).

I'd enter the following information into a file named `datafile`:

```
a = 1.2
. a, 0.6, P1
. 2, a, P2
. 3/4, 1.3, P3

! print("Rectangular coordinates in default coordinate system")
print
rotate 40
! print("\nRectangular coordinates in rotated coordinate system")
print
! print("\nPolar coordinates in rotated coordinate system")
polar
print
```

The first line shows an assignment statement, giving the variable `a` the value of 1.2. This lets you assign values to variables and use them in valid python expressions later in the datafile.

The next three command lines use the "`.`" command to enter points in Cartesian coordinates and gives them a name (note the variable `a` is used in two places to represent numbers). The `3/4` is evaluated as a python expression; the math module's symbols are in scope, so you can use expressions like `cos(pi/3)`.

The `!` command executes a single-line of python code. If you need to execute multiple python lines, use the `{` and `}` commands.

The `print` command[1] with no arguments (called the bare `print` command) prints out all the geometrical objects that have been defined in the order they were defined. Or, you can give it the names of the desired geometrical objects separated by commas and it will print their coordinates out. Here, the output is in Cartesian (rectangular) coordinates, but in two dimensions, you can also print the output in polar coordinates, as is done further down with the `polar` command.

The `rotate` command rotates the origin 40° counterclockwise about the origin. This changes the output to be in the x'y' coordinate system. Degrees are the default angle measure.

To run this datafile, I'd execute the following command in a console:

```
python xyz.py datafile
```

This causes the following results to be printed to the screen:

```
Rectangular coordinates in default coordinate system
P1 : Pt(1.200, 0.6000)
P2 : Pt(2, 1.200)
P3 : Pt(0.7500, 1.300)

Rectangular coordinates in rotated coordinate system
P1 : Pt(1.305, -0.3117)
P2 : Pt(2.303, -0.3663)
P3 : Pt(1.410, 0.5138)

Polar coordinates in rotated coordinate system
P1 : Pt<1.342, 346.6 o>
P2 : Pt<2.332, 351.0 o>
P3 : Pt<1.501, 20.02 o>
```

Let's check these results for the first point. The two-dimensional rotation matrix gives

$$\begin{bmatrix} \cos(40\,°) & \sin(40\,°) \\ -\sin(40\,°) & \cos(40\,°) \end{bmatrix} \begin{bmatrix} 1.2 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 1.3049 \\ -0.3117 \end{bmatrix}$$

This is the same as what the program printed out (highlighted in the output).

The **scale** command is convenient for a variety of uses. One use is isotropic scaling to effect a change in length units. As an example, suppose the above datafile used miles to locate the points, but we wanted the output in km. As there are 1.60934 km per mile, the use of a **scale 1.60934** command would result in the output being in units of km:

```
a = 1.2
. a, 0.6, P1
. 2, a, P2
. 3/4, 1.3, P3
scale 1.60934  # Change units from miles to km

! print("Rectangular coordinates in default coordinate system")
print
rotate 40
! print("\nRectangular coordinates in rotated coordinate system")
print
! print("\nPolar coordinates in rotated coordinate system")
polar
print
```

yields

```
Rectangular coordinates in default coordinate system
P1 : Pt(1.931, 0.9656)
P2 : Pt(3.219, 1.931)
P3 : Pt(1.207, 2.092)

Rectangular coordinates in rotated coordinate system
P1 : Pt(2.100, -0.5017)
```

1 Not the same as python's `print` command shown in the commands starting with `!`.

```
P2 : Pt(3.707, -0.5895)
P3 : Pt(2.269, 0.8268)

Polar coordinates in rotated coordinate system
P1 : Pt<2.159, 346.6 o>
P2 : Pt<3.754, 351.0 o>
P3 : Pt<2.415, 20.02 o>
```

One feature of the `xyz.py` script that I use a lot is local coordinates. Often, problems can be specified with geometrical objects in a "local" coordinate system -- i.e., coordinates that are convenient to a part of the problem at hand, but not the same as the default coordinate system.

For example, suppose I use a default coordinate system in my yard that uses a fencepost as the origin and the y-axis goes in the true north direction. Suppose I want to enter the coordinates for a point P whose location is 3.4 units west and 2.7 units south of another point Q. Further suppose Q has coordinates (75, 39) in the default coordinate system. While you can do this example in your head to get the default coordinates for point P, it can be harder for more sophisticated problems. But since the `xyz.py` script can do it easily for you, there's no need for doing the calculations manually. Here's a portion of a datafile that could be used:

```
. 75, 39, Q  # Define point Q, the new origin for the local coord. system
push
translate Q  # Change the origin to point Q
. -3.4, -2.7, P
pop          # Restore the default coordinates
print        # Show P and Q in default coordinates
```

The point Q is entered in the default coordinate system by the first line. Then the default coordinate system is "pushed" onto a stack to save the state. A transformation command `translate` is used to position the origin at point Q (i.e., define a new coordinate system). Then the coordinates of point P are entered. The basic rule is **all entered coordinates are in the current coordinate system** (at the beginning of the script, the default and current coordinate systems are the same). Internally, the script transforms these current coordinates back to the default coordinates and stores those numbers. Thus, you can imagine the points, lines, and planes you define as being rigidly located in space and the script is used to print out their coordinates in different coordinate systems of your choice.

The `pop` command "pops" the pushed coordinate system off the stack and restores it; in this case, we're back to the default coordinate system. The `print` command prints the name and coordinates of each defined object:

```
Q : Pt(75, 39)
P : Pt(71.60, 36.30)
```

If we hadn't used the `pop` command, the coordinates of Q and P would have been printed with point Q as the origin:

```
Q : Origin
P : Pt(-3.400, -2.700)
```

Note the special string `Origin` identifying the origin.

Vector algebra

Features: points, lines, and planes have vector interpretations; a useful vector algebra can be defined for them.

Points, lines, and planes can be interpreted as vectors. A point with coordinates (x, y, z) is considered the position vector $x\,\boldsymbol{i} + y\,\boldsymbol{j} + z\,\boldsymbol{k}$. A Line object is the vector between the two points $\boldsymbol{p}$ and $\boldsymbol{q}$ on the line (and these are attributes of the Line object), which is $\boldsymbol{q} - \boldsymbol{p}$ (i.e., $\boldsymbol{q}$ is the tip of the vector). A plane object is a vector made from its normal vector; since a Plane object is derived from a Line object, the Plane is already is a vector object.

An algebra is defined for these three objects and they can be combined with addition, subtraction,

dot products, and cross products. They can also be multiplied and divided by scalars.

Two forces are 6 ***i*** and 9 ***j***. What is their vector sum? We use the following datafile:

```
ijk
F1 = 6*i
F2 = 9*j
print F1 + F2
print -F1
print F1/3
print F1.dot(F2)
dot F1, F2
print F1.cross(F2)
cross F1, F2
```

The `ijk` command is a convenience command that defines the origin o, the **i**, **j**, and **k** unit vectors and the xy, xz, and yz coordinate planes. The commands in the datafile demonstrate vector addition, unary negation of a vector, dividing a vector by a scalar, and the dot and cross products (two different ways of calculating them).

Running this datafile results in

```
F1 + F2 : Ln(Origin, Pt(6, 9))
-F1 : Ln(Pt(6, 0), Origin)
F1/3 : Ln(Origin, Pt(2, 0))
F1.dot(F2) = 0
F1 dot F2 = 0
F1.cross(F2) : Ln(Origin, Pt(0, 0, 54))
cross Ln(Origin, Pt(6, 0)) X Ln(Origin, Pt(0, 9)):  Ln(Origin, Pt(0, 0, 54))
```

The short-hand `Ln` and `Pt` denote lines and points, respectively (planes are `Pl`). The last two lines show that the cross product extends out of the xy plane because all three of its points' coordinates are shown. The `xyz.py` script is set up by default to display points in the xy plane with two coordinates. The third spatial coordinate will only be shown if the point or line don't lie in the xy plane (use `2D off` to change this behavior if you don't want it). This makes it handy for two-dimensional analytical geometry calculations.

The following demonstrates that points, lines, and planes all behave as vectors. Calculate the vector sum of the two vectors a = (1, 0) and b = (0, 1) by using points:

```
. 1, 0, a
. 0, 1, b
f = a + b
print f
```

which gives the vector $\boldsymbol{i} + \boldsymbol{j}$:

```
f : Ln(Origin, Pt(1, 1))
```

You can get the same result by adding the Cartesian unit vectors:

```
ijk
f = i + j
print f
```

which gives

```
f : Ln(Origin, Pt(1, 1))
```

A third way is to add the xz and yz planes (which have the ***j*** and ***i*** normal vectors, respectively):

```
ijk
f = xz + yz
print f
```

which gives

```
f : Pl(Origin, Pt(1, 1, 0))
```

Note when a Plane object is printed, its z coordinate is always shown, regardless of the `2D` setting.

These three different methods give the same results (vectorially, although the types are different) because, behind the scenes, they are vectorially adding the same objects.

This algebra gives useful semantics for getting other objects. For example, you can add a vector to a plane to get another plane: `i + yz` gives a plane that is perpendicular to the xy plane and intersects both the x and y axes at 45°.

Locating something underground

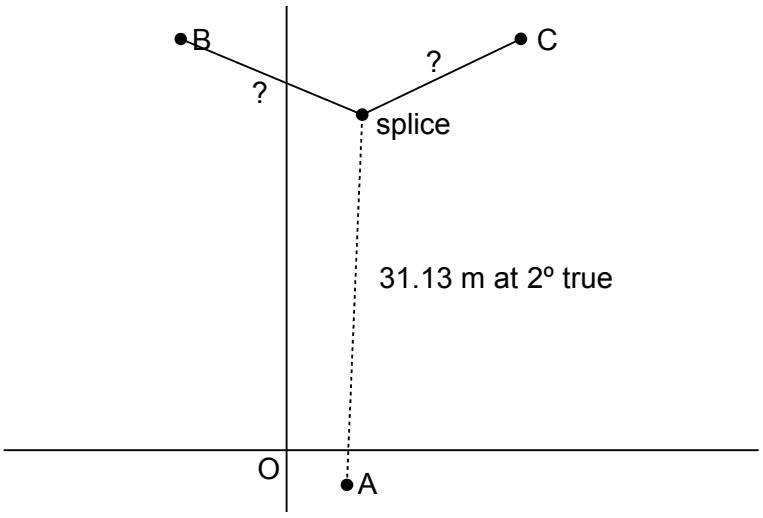
Features: calculating coordinate transformations with measurement uncertainty, compass mode.

This is a made-up example, but it is similar to a problem I had to solve last summer in my yard. The datafile is given as `xyz.splice`.

Suppose I have an electrical cable underground that runs from my house to some buried sprinkler valves in the yard. The valves are not working correctly and I suspect a buried splice in the control wires has gone bad (resistance measurements from both ends indicate a problem). I need to dig the splice up, check it, and fix it if it's bad. Thus, I need to figure out where to dig to expose the splice.

In the following, I include uncertainty estimates of the measurements in the usual short form notation: 34.7(3) is interpreted as the mean of the measurement is 34.7 and the standard deviation is 0.3. One of the strengths of the `xyz.py` script is that it can propagate the input uncertainties into the calculated results (this is due to the abilities of the python uncertainties library [punc]).

My notes indicate the splice has a true compass bearing of 2° from point A and is at a distance of 31.13(5) m from point A. Point A, in turn, is on a bearing of 130° from point O and 3.3 m from it; I use point O as the origin of coordinates in the back yard, as it is the corner of a large cement pad. Here's a picture:



Because of things interfering in the landscape, I can't measure directly from point A or O to locate the splice. Instead, I need to measure from two reference points B and C (a telephone pole and a fence post at the north edge of my property[2]). **The objective is to calculate the distances from points B and C to the splice**; these numbers will tell me where to dig.

The distances and compass bearings of these points with their estimated uncertainties are

2 These measurements were made years ago before the intervening landscape objects were in place.

Point	Distance, m, from point O	Bearing, °
A	3.30(1)	130(2)
B	35.77(1)	349.3(5)
C	38.85(10)	25.6(5)

Points A and B were measured with a laser distance meter and have an estimated distance uncertainty of 10 mm. Point C was measured with a tape measure and the distance uncertainty is estimated at 100 mm. The bearing for point A was measured with a small hand-held magnetic compass and the uncertainty is estimated at 2°. For points B and C, the bearings were measured from a satellite photo and the uncertainty is estimated at 0.5°.

Given this information, I want to use the `xyz.py` script to calculate the distances to the splice from points B and C, as I can readily measure from them with a tape measure because there are no intervening objects. An advantage of estimating the uncertainties of the measurements is that they will be propagated into the resultant distance measurements, giving me a feel for how closely the location of the splice is computed. This is important, as the ground is hard and it's a lot of work to dig down to locate the wire[3].

Here's the datafile to solve this problem (in the file `xyz.splice`):

```
# Locating an underground splice

deg             # Degrees is the default, but it doesn't hurt to show
                # it explicitly.
compass on      # Angles measured clockwise from true north
indent 2        # Indent output two spaces
polar           # Show output in polar coordinates

# The three points in the problem
<  3.300(1), 130(2), A
< 35.77(1),  349.3(5), B
< 38.85(10),  25.6(5), C

# Translate the origin to point A so we can enter the splice location
push                        # Save current coordinate system
translate A
< 31.13(5), 2(2), splice    # Note we're entering in polar coordinates
! out("Verification of splice point with respect to point A")
print splice                # So we can verify it's correct
pop                         # Restore default coordinate system

! out("Verification of other points with respect to origin O")
print A, B, C  # Print all the defined points in order defined

# Print desired distances
! out("\nDistances in m")
dist B, splice
dist C, splice
```

The `print` command is important, as it lets you verify that you entered the input data correctly. The results are

```
Verification of splice point with respect to point A
  splice : Pt<31.13(5), 2(2) oC>
Verification of other points with respect to origin O
  A : Pt<3.300(1), 130(2) oC>
  B : Pt<35.77(1), 349.3(5) oC>
  C : Pt<38.9(1), 25.6(5) oC>

Distances in m
  Distance: B to splice = 12.0(1)
```

3 When I have to do such digging, I'll run a portable sprinkler over the area for 1-2 hours per day for a few days and the absorbed water makes the ground easier to dig.

```
Distance: C to splice = 14(1)
```

The angle brackets `<...>` indicate polar coordinates and the `oC` means the angles are being measured in degrees (`o`) and compass mode is on (`C`, i.e., angles are being measured clockwise from north).

You can see that the measurement uncertainty for the distance from point C is substantially larger than from point B. Reducing the uncertainty of the point C location was my first intuition for better location of the splice, but if you change the standard deviation of 10 to 1, it makes no difference. The uncertainty then probably comes from the angle uncertainty of the splice. Reducing the angle uncertainty from 2 degrees to 0.2 degrees gives a splice to C distance with the same uncertainty as the splice to B distance. Unfortunately, I can't improve on that angle measurement until I dig the splice up! Actually, what I'll do when I locate the splice is update the distance measurements from points B and C and use them to locate the splice instead of using the compass bearing.

Two-dimensional geometry problems like this are why I wrote the `xyz.py` script, as I need such things occasionally. While it's nice that the script also handles three-dimensional problems, I find I use it more for the two-dimensional problems.

One of the most important uses of the datafile is that I can put its contents into a document, save the document in a revision control system, then have that information later when I want to understand what I did. Before we had computers and digital cameras to record things, I'd take pictures of things and write down dimensions, but those things have a tendency of getting lost or separated over time. The digital information stays on the computer and I know right where it is.

Another category of use of the `xyz.py` script is the reduction of some measurements a machinist might make of an object on a surface plate clamped to an angle plate. I won't give an example, as the basic principles are the same as those demonstrated in the above splice location example. The idea is that you clamp the work to an angle plate, then use a height gauge to measure the distance of the work's features above the plane of a surface plate. You then rotate the work/angle plate by a right angle about an axis parallel to the plane of the surface plate and measure the features again. This gives you the x and y coordinates of the features referenced to the surface plate. You usually want to have things referenced to another more convenient coordinate system (say, located at a particular feature) and you want to calculate distances and angles between the features; the `xyz.py` script was designed to handle such things.

Area calculation

Features: area of a planar polygon.

The script is capable of calculating the area of an arbitrary polygon of points **in the xy plane**. To be meaningful, the polygon must not be self-intersecting. The method used is the "shoelace algorithm", which makes sense once you see a picture of the method.

A polygon was defined in the file `xy.area`:

```
# This datafile is a test case that was generated by drawing a polygon
# on paper.  The area of the polygon was calculated by triangulating
# it and measuring the sides of each triangle.  These side lengths
# were verified by using the dist commands given below.

indent 2

!out("Test case for area calculation\n")

.  15.5, 41.5, a
.  70.2, 15.9, b
. 130.4, 26.9, c
. 164.0, 88.3, d
.  83.5, 88.1, e
.  68.8, 59.6, f

!out("Lengths of polygon's sides:")
```

```
dist a, b
dist b, c
dist c, d
dist d, e
dist e, f
dist f, a

!out("\nLengths of internal triangles' sides:")
dist b, f
dist c, f
dist c, e

# The independent method of calculating the area was within 0.01% of
# the calculated value by the xy.py script.
area a, b, c, d, e, f
!out("\nExpect area to be 6086")

# If you e.g. exchange points c and f, you'll get the warning message
# that the polygon is self-intersecting.
```

This was used to validate the area calculation and the self-intersection algorithm. The results are

```
Test case for area calculation

Lengths of polygon's sides:
  Distance: a to b = 60.39
  Distance: b to c = 61.20
  Distance: c to d = 69.99
  Distance: d to e = 80.50
  Distance: e to f = 32.07
  Distance: f to a = 56.29

Lengths of internal triangles' sides:
  Distance: b to f = 43.72
  Distance: c to f = 69.74
  Distance: c to e = 77.10
  Area for points:  a, b, c, d, e, f
    Area = 6086

Expect area to be 6086
```

As noted in the comments, the answer was within 0.01% of an independently-computed value from triangulation.

Reflection

Features: reflection about points and lines.

The `xyz.py` script has the ability to reflect points and lines about points, lines (in the xy plane only), and planes. Here, we look at a simple use of this to define some points. We'll create four points that are the corners of the unit square. We'll create only one point, then use reflections to create the other three points. Here's the datafile:

```
# Example of using reflections
ijk

# Make the first point
. 1, 1, a

# Make a copy, then reflect it about the origin
b = a.copy
reflect o, b    # o is a point, the origin

# Reflect b about the x axis to get the third point
c = b.copy
reflect i, c    # i is a unit vector (in reality, a line)

# Reflect b about the y axis to get the fourth point
```

```
d = b.copy
reflect j, d

# Show the four points
print a, b, c, d
```

which results in

```
Reflected about Origin:
  b: Pt(1, 1) --> Pt(-1, -1)
Reflected about Ln(Origin, Pt(1, 0)):
  c: Pt(-1, -1) --> Pt(-1, 1)
Reflected about Ln(Origin, Pt(0, 1)):
  d: Pt(-1, -1) --> Pt(1, -1)
a : Pt(1, 1)
b : Pt(-1, -1)
c : Pt(-1, 1)
d : Pt(1, -1)
```

The `ijk` command is a convenience command to make the origin, Cartesian unit vectors, and coordinate planes objects. It's important to make a copy of an existing point because the `reflect` command changes the coordinates of the point(s) it operates on.

The ability to perform translation and rotation transformations allows a rather simple implementation of generating reflections. Reflection about a point is easy: translate the coordinate system to the point and then change the sign of all the projected objects' coordinates. Reflection in a plane is also straightforward: translate the origin to a point in the plane, then rotate the coordinate system so that the plane becomes the xy plane. Then negate the z coordinate of each point (or points defining a line or plane). Reflection about a line is, in reality, just reflection about a plane perpendicular to the xy plane that contains the line.

Centroid

Features: calculating the total mass and centroid of a group of particles with mass.

Points can be defined to include mass; when you define the point, include the keyword **m** to define the mass. When the **centroid** command is used, the indicated points (or all of them if no point arguments were given) will have their centroid and total mass calculated.

Here's a simple example. Suppose we have unit masses at the corners of a unit square with one of the masses being twice the others. If the square is centered around the origin, then the total mass will be 5 and the centroid will be slightly offset from the origin:

```
p = 1
M = 1
.  p,  p, a, m=2*M
. -p,  p, b, m=M
. -p, -p, c, m=M
.  p, -p, d, m=M
centroid
# Show the information is saved in two variables
!print "Total mass", total_mass
!print "Centroid", centroid
```

results in the output

```
Centroid:  mass = 5 at Pt(0.2000, 0.2000)
Total mass 5
Centroid Pt(0.2000, 0.2000)
```

The first two lines of this datafile show that you can use variable assignments and the third line shows that expressions can be used (the mass `M` is multiplied by 2).

Because you might want to use the calculated values of the total mass and the centroid point later in a script, they are saved in the variables `total_mass` and `centroid`.

Bolt circles

Features: changing an object's position in space with object attributes, in-line python code.

In the shop, sometimes we need to lay out a pattern for a bolt circle. A bolt circle is the location of holes for bolts that fall on a circle, such as for a pipe flange. In the days before CNC machines, machinists would sometimes lay out such patterns using Cartesian coordinates and do precise location of features using this method with a jig boring machine. *Machinery's Handbook* contains a table to help you do such things (but it's straightforward to do with the Cartesian equation of a circle and a calculator).

Here, we'll show the calculated points of a bolt circle of 6 holes, equally spaced. The example shows the utility of defining a point in the datafile, then using a chunk of python code to iterate to get the solution.

Suppose the bolt circle diameter is 200 and we want the six holes to start at a polar angle of 0° counterclockwise from the x axis. We define the starting point in polar coordinates, then increment the polar angle to get the needed points. In contrast to the other examples in this document, this example uses a fixed coordinate system and moves the point around in space by changing its polar azimuth angle.

```
< 100, 0, a
deg     # Make sure we're using degrees
#polar
#eps 5e-14
{
out(a)                      # Show the first point
for i in range(5):          # Calculate the other five points
    a.theta += 60
    out(a)
}
```

Arbitrary python code can be included between the curly braces. For example, you could define functions, classes, and variables that you need to solve the problem you're working on. Running `xyz.py` with this datafile produces

```
Pt(100, 0)
Pt(50.00, 86.60)
Pt(-50.00, 86.60)
Pt(-100.0, 1.225e-14)
Pt(-50.00, -86.60)
Pt(50.00, -86.60)
```

If you uncomment the `#polar` command in the datafile and run it again, you'll see that the points are positioned as you would expect. If you uncomment the `eps` line, the roundoff error in the fourth point will be set to zero.

Three-dimensional examples

Great circle distance

Features: spherical coordinates, elevation mode.

Suppose we want to calculate the straight line and great circle distance between San Francisco, USA and New Delhi, India. The `xyz.py` script can do this for us.

We'll assume the Earth is a sphere with a radius of 6371 km. San Francisco is at 37°47′ N 122°25′ W and New Delhi is at 28°37′N 77°13′E.

The following datafile will solve the problem (python expressions are handy to convert the sexigesimal measure):

```
r = 6371  # Radius of Earth in km
elev on   # Use elevation angles (psi = pi/2 - phi) instead of phi
indent 2  # Indent the output by 2 spaces
```

```
# Define the location of the two cities in spherical coordinates r,
# theta, and psi (elevation angle = latitude).
<< r, -(122 + 25/60), 37 + 47/60, SF
<< r,    77 + 13/60,  28 + 37/60, ND

digits 6
dist SF, ND, dist
angle SF, ND, ang

{
if 1:
    out("Results:")
    out("  Chordal distance =", sig(dist))
    out("  Angle =", sig(ang), "deg =", sig(ang*pi/180), "rad")
    # Great circle distance = arc length = radius*angle
    out("  Great circle distance (km) =", sig(r*ang*pi/180))
}

! out("\nCartesian coordinates of cities:")
print SF, ND
! out("\nSpherical coordinates of cities:")
print SF, ND, <<
```

I used the `if 1:` to allow me to indent the python code, which makes it a little easier to read (`sig` is a convenience function to convert a number to a string with a specified number of significant figures). The results are

```
Results:
  Chordal distance = 10507.4
  Angle = 111.101 deg = 1.93908 rad
  Great circle distance (km) = 12353.9

Cartesian coordinates of cities:
  SF : Pt(-2699.24, -4250.59, 3903.37)
  ND : Pt(1237.48, 5454.12, 3051.37)

Spherical coordinates of cities:
  SF : Pt<<6371.00, 237.583, 37.7833 oE>>
  ND : Pt<<6371.00, 77.2167, 28.6167 oE>>
```

We turned elevation mode on so we could use the latitude numbers directly. You can use the usual spherical coordinate ϕ and get the same results by changing the lines:

```
elev off
<< r, -(122 + 25/60), 90 - (37 + 47/60), SF
<< r,    77 + 13/60,  90 - (28 + 37/60), ND
```

where the ϕ angle is defined by an expression (i.e., subtracting ψ from 90°). The ability to use expressions increases the utility of the script.

The last line uses the slightly screwy syntax of making `<<` one of the "objects" to print; this is an instruction to turn on spherical coordinates for this print command, overriding the system that is currently set. It is useful to print out the input data, as it can help you verify that you've entered the data correctly and aren't getting the right answer to the wrong problem.

The great circle distance can be checked with spherical trigonometry. Using the indicated web page's notation:

$$\begin{aligned} &\text{Point A (New Delhi):} && \lambda_A = 37.78,\ L_A = 77.22 \\ &\text{Point B (San Francisco):} && \lambda_B = 28.62,\ L_B = -122.42 \end{aligned}$$

Plugging into the formula, we get

$$\begin{aligned} \text{arc length} &= r\cos^{-1}\left[\sin\lambda_A\sin\lambda_B + \cos\lambda_B\cos\lambda_B\cos(L_A - L_B)\right] \\ &= 6371\cos^{-1}\left(\sin 37.78\,^\circ\sin 28.62\,^\circ + \cos 37.78\,^\circ\cos 28.62\,^\circ\cos 199.63\,^\circ\right) \\ &= 12353.96 \end{aligned}$$

(remember to get the inverse cosine in radians). We can check the chordal distance by using the Cartesian coordinates:

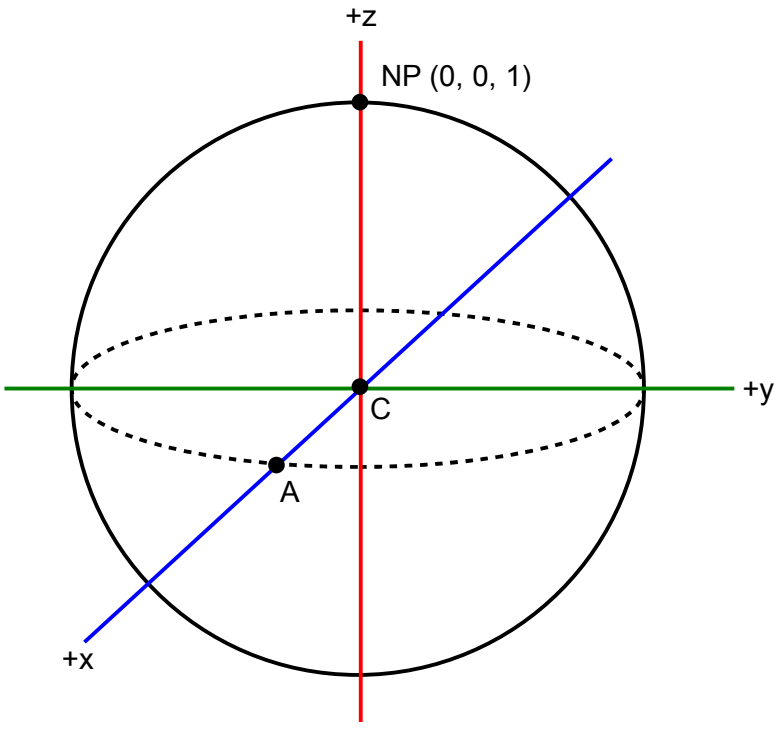
$$\text{chordal distance} = \sqrt{(-2699.24-1237.48)^2+(-4250.59-5454.12)^2+(3903.37-3051.37)^2}$$
$$= 10507.38$$

Rotations

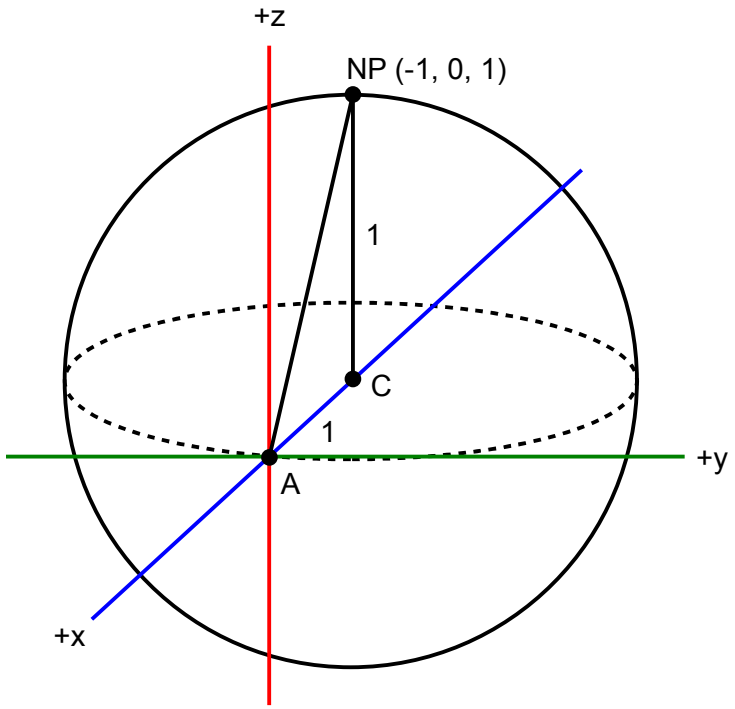
Features: rotation transformations, nicely-formatted printing.

I've found that I use rotations a lot in three dimensional problems, as they can make many problems simpler to solve. But my chronologically-gifted mind can sometimes be resistant to imagining rotations and translations in three-dimensional space -- or at least more resistant than when I was younger. Here, I give an example of transforming coordinates on a unit sphere that are relatively easy to visualize (and can give you more confidence that you can see what's going on). The problem involves a translation and two rotations (and they are similar to what's used in the **ISS visibility** below. The rotations are by a right angle each time, making it easier to see the results by inspection of the numbers. When rotating about an axis, remember that when you look at the axis' unit vector, the tip of the vector pokes you in the eye and the rotations are counterclockwise around the rotation axis. I'll use a conventional right-hand coordinate system.

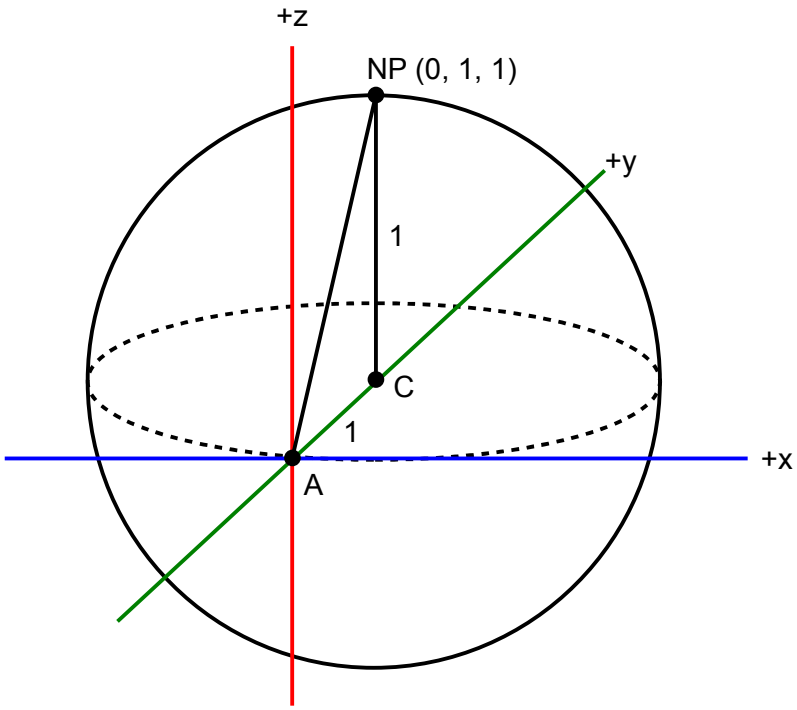
The first step is to make the unit sphere's center C the origin and put the north pole at (0, 0, 1). Define the point A at (1, 0, 0) on the equator. We'll make the original x, y, and z axes blue, green, and red, respectively, to show how they change orientation after the transformations. Angles are in degrees.



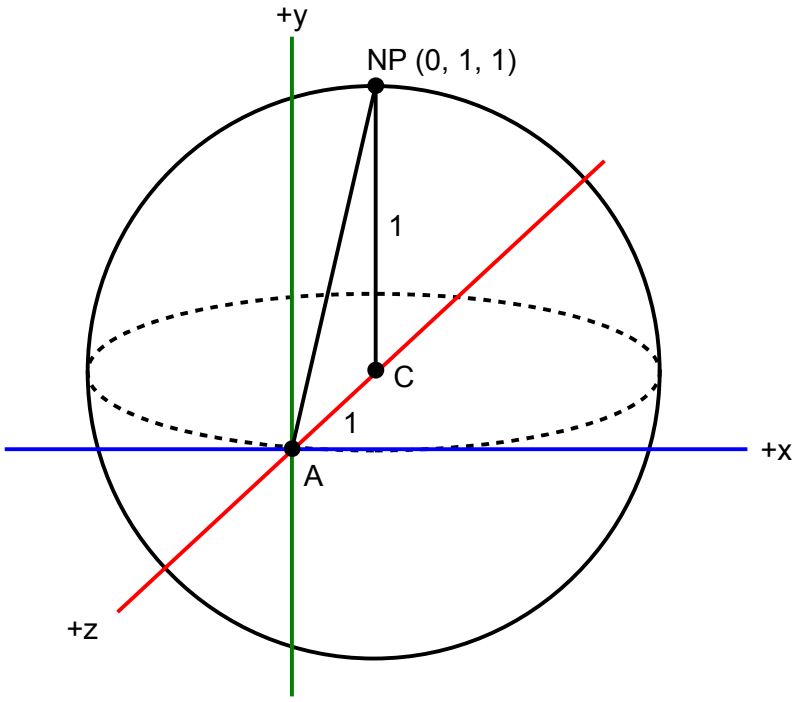
Translate the origin to point A. The north pole should now have the coordinates <<$\sqrt{2}$, 180, 45>> in spherical coordinates and (-1, 0, 1) in Cartesian coordinates:



Define **i**, **j**, **k** in the current system and rotate 90° about the z axis. This will make the x axis point east along the equator. The north pole should be at $<<\sqrt{2},\ 90,\ 45>>$ or (0, 1, 1):



Finally, define **i**, **j**, **k** in the current system and rotate 90° about the x axis. This will make the z axis normal to the sphere's surface and make the xy plane the tangent plane touching the sphere at point A. The north pole should be at $<<\sqrt{2},\ 90,\ 135>>$, which is (0, 1, -1).



Here's a datafile which performs these operations (the datafile is the `xyz.sphere` file):

```
# Rotations on a unit sphere (see xyz.pdf).

2D off
indent 4

# 1.  Define the points
    . 0, 0, 1, np   # North pole
    . 1, 0, 0, A    # Point at equator
    sph
    !out("1.  Before rotation")
    print np, A
    rect
    print np, A
    sph

# 2.  Translation
    translate A
    !out("\n2.  After translation")
    print np
    rect
    print np
    sph
    # x axis is pointing in radial direction, y axis is pointing east

# 3.  Rotate 90 deg about z axis to make x axis point east along equator.
    ijk
    rotate 90, k
    !out("\n3.  After 90 deg rot about z axis")
    print np
    rect
    print np
    sph

# 4.  Rotate 90 deg about x axis to make z axis point in radial direction.
    ijk
    rotate 90, i
    # xy plane now tangent to sphere
    !out("\n4.  After 90 deg rot about x axis")
    print np
```

```
rect
print np
```

The results are

```
1.  Before rotation
    np : Pt<<1, 0, 0 o>>
    A  : Pt<<1, 0, 90 o>>
    np : Pt(0, 0, 1)
    A  : Pt(1, 0, 0)

2.  After translation
    np : Pt<<1.414, 180, 45 o>>
    np : Pt(-1, 0, 1)

3.  After 90 deg rot about z axis
    np : Pt<<1.414, 90, 45 o>>
    np : Pt(0, 1, 1)

4.  After 90 deg rot about x axis
    np : Pt<<1.414, 90, 135 o>>
    np : Pt(0, 1, -1)
```

IMPORTANT: when performing multiple rotations like these, it's important to redefine the **i**, **j**, **k** vectors with the `ijk` command in the new coordinate system. If you don't, you'll use their values in the old coordinate system and you probably won't get the results you expect.

ISS visibility

Features: rotations, translations.

Suppose I'm in San Francisco (37°47′ N 122°25′ W) and **I want to know if I can see the International Space Station (ISS)**. The ISS is in an orbit from 404-424 km above mean sea level and orbit inclination is 51.6° (the orbital plane's angle with respect to the plane of the Earth's equator). To be specific, suppose the ISS is at an altitude of 415 km and is directly over Salt Lake City, UT (40°45′ N 111°53′ W). Earth's equatorial radius is 6378.14 km; we'll assume the Earth is a sphere. This datafile is included in the package under the name `xyz.iss`.

To solve this problem, we'll locate San Francisco and the ISS as points in space; the coordinate system will have its origin at the Earth's center and the north pole will be on the +z axis. Then we'll translate the coordinate system to San Francisco's location on the Earth and rotate things so that we're in San Francisco's topocentric coordinate system. This means the xy plane is tangent to the sphere at San Francisco. The ISS will theoretically be visible if the elevation angle is greater than zero[4]. This problem is analogous to the problem in section **Rotations** above, but doesn't involve right angles, so it can be harder to visualize.

Here's the datafile:

```
# Is the ISS visible from San Francisco when it is over Salt Lake City?

# Define variables
r = 6378.14     # Radius of Earth in km
alt = 415       # Altitude in km of ISS above mean sea level

# Define the north pole point as a check at the end to see that the
# coordinates in SF's topocentric coordinates make sense.
. 0, 0, r, np
sph
! out("North pole at start:")
print np

# Latitude and longitude of San Francisco.  We'll use SF's longitude
# as the zero angle and SLC's will be a positive polar theta angle
```

4 Things like building obstructions, light pollution, and time of day would also be important to visibility, but we're just looking at the geometric problem.

```
# from SF when the Earth's center is the origin and the north pole
# points up the +z axis.
sf_phi  = 90 - (37 + 47/60)
sf_long = 0

# Latitude and longitude of Salt Lake City converted to spherical phi
# angles
slc_phi  = 90 - (40 + 45/60)
slc_long = (122 + 25/60) - (111 + 53/60)

! out("Longitude difference between SF and SLC =", sig(slc_long), "deg")

# Define points
<< r      , sf_long,  sf_phi,  sf
<< r      , slc_long, slc_phi, slc
<< r + alt, slc_long, slc_phi, iss
! out("\nOther points defined:")
print sf, slc, iss

# We can make an estimate of the altitude angle of the ISS using an
# approximate right triangle whose base is the chordal distance
# distance between SF and SLC and the triangle's height is the
# height above sea level of the ISS.  Note this will overestimate the
# true elevation because the Earth curves under SF's tangent plane.
dist sf, slc, dist
elev_estimate = atan(alt/dist)*180/pi
! out("\nChordal distance =", sig(dist), "km")
! out("Elevation estimate =", sig(elev_estimate), "deg")

# Translate the origin to San Francisco
translate sf

# Rotate about the y axis by the SF's phi to make the z axis be normal
# to SF's tangent plane.  Note we use the ijk command to get the unit
# vectors in the translated system.
ijk
rotate sf_phi, j

# Change to spherical coordinates and print out the ISS' coordinates
# in SF's topocentric coordinate system.  The third coordinate will be
# the elevation above or below the tangent plane.
! out("\nISS coordinates in SF's topocentric system (elevation mode on)")
elev on
print iss
! out("ISS elevation from horizon =", sig(90 - iss.phi*180/pi), "deg")
```

and here are the results:

```
North pole at start:
  np : Pt<<6378, 0, 0 o>>
Longitude difference between SF and SLC = 10.53 deg

Point definitions:
  sf  : Pt<<6378, 0, 52.22 o>>
  slc : Pt<<6378, 10.53, 49.25 o>>
  iss : Pt<<6793, 10.53, 49.25 o>>

Chordal distance = 964.3 km
Elevation estimate = 23.28 deg
North pole to SF distance = 5614 km

ISS coordinates in SF's topocentric system (elevation mode on)
  iss : Pt<<1078, 113.3, 18.23 oE>>

North pole in SF's topocentric coordinates
  np : Pt<<5614, 180, -26.11 oE>>
```

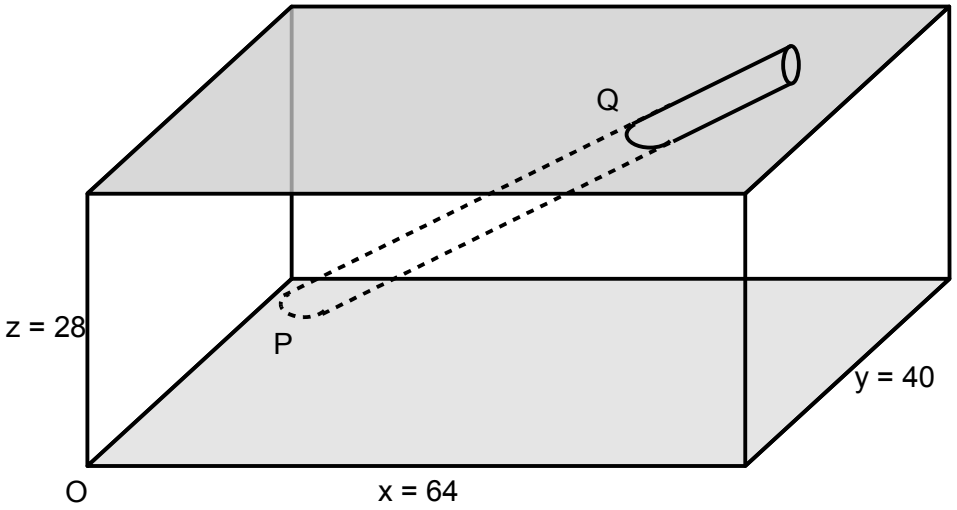
You can see that the ISS is 18.23° above the horizon in San Francisco, so it's theoretically visible.

Also note that the north pole's coordinates in San Francisco's topocentric coordinates make sense -- the radial distance was calculated to be 5614 km and the azimuthal direction is 180° (i.e., in the -x direction). This is easier to see if you use your right hand in the usual right angle configuration with your thumb pointing north; rotate about the middle finger (y axis) and see where the north pole is in relation to the rotated system. Note the north pole is below San Francisco's horizon (as you'd expect) because the rotation angle was more than 45 deg.

Shop problem

Features: traces (projection angles), orthogonal projection, angles between objects, intersections between objects, direction cosines and direction angles.

I have a rectangular box that I must drill a hole into and pass a pipe through the hole to exit at a proper location in the bottom. Because I want the penetration on the top to be esthetically pleasing and because there's a lack of clearance, I need to drill the entrance and exit holes carefully so that the pipe fits snugly in the holes. Here's a picture:



Suppose I locate a Cartesian coordinate system at O and the axes and dimensions of the box are as indicated in the figure (dimensions in inches). Further suppose that I need the centerline of the pipe to enter at point P with coordinates P(5, 36, 0) and exit at point Q with coordinates Q(55, 5, 28).

The pipe is 1 inch nominal US pipe with an outside diameter of 1.315. I'll use a 1-5/16 Forstner bit to drill the holes for a close fit (there will be some filing/sanding). The problem is: **how do I set up the drilling process so that the holes come out in the correct positions and at the correct angles?**

The task is to locate a line in space, then calculating its projection angles (i.e., traces) or its azimuth and elevation with respect to the two coordinate planes xz and yz. Here's the datafile (it's `xyz.shop`):

```
# Shop problem example

2D off  # Show all three coordinates of points
indent 2

# Define points and line
. 5, 36, P
. 55, 5, 28, Q
| P, Q, line

ijk     # We need the xy plane

# Define the top plane of the box with three noncollinear points
t = 28
. 0, 0, t, q1
. 1, 1, t, q2
```

```
. 0, 1, t, q3
plane q1, q2, q3, top

! out("Verify setup:")
print P, Q, top

# Calculate the intersection points to verify the line is correct
! out("\nIntersections to verify correct line and planes:")
intersect xy, line
intersect top, line

# Show the trace for point Q (projection angles)
! out("\nTraces (projection angles) for point Q:")
trace Q

# Show the spherical coordinates for P and Q
! out("\nSpherical coordinates for the points:")
print P, Q, <<

indent 0

# Show the angle between the line and the xy plane
! out()
angle xy, line
! out()

# Project the two points P and Q into the xy plane and display the
# direction cosines and angles of the resulting line.  Note we make
# copies of P and Q because the project command changes its arguments.
q = Q.copy
p = P.copy
project xy, p, q
| p, q, ln
! out("\nDirection cosines and angles of projected line:")
indent 2
dc ln
```

While some of the things being printed are for pedagogical reasons, it's still a good idea to print such information so that you can check that you've entered the problem's data correctly.

First is the definition of the points P and Q and the top and xy planes. The `ijk` command is a convenience command for getting the Cartesian unit vectors, origin, and coordinate planes. The `intersect` commands demonstrate that the defined `line` intersects the top and xy planes where you'd expect them to; this verifies that things have been set up correctly.

The `trace` command prints the angles between the orthogonal projections of the point Q onto the coordinate planes. These trace angles are useful for shop measurements with sine sticks (see [an] or `sine_sticks.pdf` at http://code.google.com/p/hobbyutil/). If you know these trace angles, you can calculate the spherical coordinates θ and ϕ for a unit vector lying on a line (and vice versa). This is useful in dealing with e.g. drilling holes with axes that are not in the coordinate planes (i.e., the problem we're dealing with here).

The `sph` command changes the output coordinate system to spherical coordinates and `print Q` prints the point `Q`'s spherical coordinates (flagged by the double angle brackets; the `o` means the angles are measured in degrees).

The `angle` command gives the angle between the line and top plane. This is the angle you'd raise the axis of the drill out of the top plane.

The `project` command orthogonally projects the points `p` and `q` (copies of `P` and `Q`) into the xy plane. These projected points can then be used to create a line in the xy plane. The `dc line` prints out the direction cosines and direction angles of this line.

Here are the results when the script is run with this datafile (note the diagnostic information confirms we entered the problem's numbers correctly):

```
Verify setup:
  P   : Pt(5, 36, 0)
  Q   : Pt(55, 5, 28)
  top : Pl(Pt(0, 0, 28), Pt(0, 0, 29))

Intersections to verify correct line and planes:
  intersect xy, line:  Pt(5, 36, 0)
  intersect top, line:  Pt(55, 5, 28)

Traces (projection angles) for point Q:
  trace Q[Pt(55, 5, 28)]:  x: 26.98, y: 79.88, z: 26.89 deg

Spherical coordinates for the points:
  P : Pt<<36.35, 82.09, 90 o>>
  Q : Pt<<61.92, 5.194, 63.11 o>>

Angle between xy and line = 25.45 deg

Projection xy[Pl(Origin, Pt(0, 0, 1))] (orthogonal)
  p: Pt(5, 36, 0) --> Pt(5, 36, 0)
  q: Pt(55, 5, 28) --> Pt(55, 5, 0)

Direction cosines and angles of projected line:
  ln (Ln):  dc = (0.8499, -0.5269, 0.000), dir ang = (31.80, 121.8, 90.00) deg
```

I've highlighted the two important angles in the output. The traces could be used with sine sticks [an] and a protruding piece of pipe to verify things were drilled at the correct angles.

If you were doing approximate work, you could draw a line on the top through point Q's intersection with the top and angle it 31.8° from the x axis. Looking down from the top, you'd position the drill bit's axis along this line and raise the drill 25.45° above the top plane to drill the hole.

For the work we want to do here, we'd want to make a drilling fixture with a hole at the correct angle to guide the drill bit if we were drilling with a hand-held drill. This fixture's guide hole would be drilled on a drill press or milling machine with a suitable fixture to hold it at the correct angles.

While I'd agree that this problem could be solved with pencil and paper, an advantage of doing it with the `xyz.py` script is that you can save the datafile and the results and thus document your design. This is often a good idea, as you later want to duplicate your work or make a modification. It's easy to change the datafile and get new results.

Cross product in left-handed coordinate system

Features: vector cross product.

The cross product in a left-handed coordinate system uses the left-hand rule to get the direction of the cross product (this is built-in to the cross product definition in Cartesian coordinates). The behavior is shown by the following datafile:

```
# Check that cross product works as expected for a left-handed system
indent 2

# Show i X j in original system
    ijk
    cross i, j, a
    ! out("i X j in original system:")
    print a

# Show i X j in left-handed system
    push
    scale -1, 1, 1
    cross i, j, a
    ! out("\ni X j in left-handed system:")
    print a

# Redefine i X j in left-handed system and calculate new cross product
    ijk
```

```
    cross i, j, a
    ! out("\nNewly-defined i X j in left-handed system:")
    print a

# Print left-handed i X j in default right-handed system
    pop
    cross i, j, a
    ! out("\nNewly-defined i X j in original right-handed system:")
    print a
```

yields

```
i X j in original system:
  a : Ln(Pt(0, 0, 0), Pt(0, 0, 1))

i X j in left-handed system:
  a : Ln(Pt(0, 0, 0), Pt(0, 0, -1))

Newly-defined i X j in left-handed system:
  a : Ln(Pt(0, 0, 0), Pt(0, 0, 1))

Newly-defined i X j in original right-handed system:
  a : Ln(Pt(0, 0, 0), Pt(0, 0, -1))
```

The `ijk` command defines the Cartesian unit vectors as (1, 0, 0), (0, 1, 0), and (0, 0, 1). The first cross product gives the expected result $\boldsymbol{i} \times \boldsymbol{j} = \boldsymbol{k}$. Then the following paragraph switches to a left-handed coordinate system. If you make a sketch, you'll see this second vector is correctly gotten by using the left-hand rule in the left-handed system. The end result is that the unit vectors in the original system still give the same resulting vectors in space -- even though the components are of course different.

The remaining two paragraphs of code show the same behavior when using the `ijk` command in the left-handed system.

The cross product is a pseudovector, which means it changes to the opposite direction when you change the handedness of the coordinate system. This is important when it represents a physical quantity because you want the vector to represent the behavior of the real physical quantity. An example is when you are sitting at the origin of a Cartesian coordinate system and you view a car driving away from you along the +x axis. In a right-handed coordinate system, the angular momentum vector of a wheel points in the +y direction. If you switch to a left-handed coordinate system, this angular momentum vector still needs to point in the same physical direction (i.e., the physical nature of the angular momentum hasn't changed). That means it must be a pseudovector, which means it changes to the opposite direction when the handedness of the coordinate system changes and thus it points along the -y direction in the left-handed system. Since angular momentum is a length vector and a linear momentum vector multiplied together using the cross product, the result is a pseudovector as needed.

The cross product is only defined in three-dimensional space. This is useful but clumsy. More modern treatments use the outer product [ga] which allow "cross products" to be defined in spaces of any dimension. But this isn't relevant for the `xyz.py` script, as the script only applies to two- and three-dimensional Euclidean spaces.

In-line code

If you're a python programmer, you may find that the ability to insert in-line code lets you do something more easily. Here's an example that defines a point, has some in-line code create a numpy array, print the point's information, then plots the square of the array's data:

```
# Define a point
. 1, 2, 3, a

# Set up an array
{
from pylab import *
```

```
x = arange(0, abs(a), .1)
}

# Print a geometrical object
print a

# Plot the previously-defined array
{
plot(x, x*x)
grid()
show()
}
```

You'll need numpy and matplotlib installed for this code to work. I won't bother showing the output, as it's straightforward to see it if you have those tools.

For program development, a disadvantage of this approach is that it can be difficult to debug your in-line code. You can start the debugger and step through the code; you just can't see the lines of code as they execute (there may be some way to do this, but I'm not aware of it).

Commands

This section summarizes the syntax of the script's commands.

All variable and object names must be valid python identifiers, meaning they must begin with a letter or underscore, then have an arbitrary number of digits, underscores, and letters following.

In the following, **pt** denotes a point, **ln** denotes a line, **pl** denotes a plane, and **o** denotes an object that can be a point, line, or plane. Keywords to a command are shown in **this** font and are of the form **keyword**=value where value can be a number, expression, or string.

Leading whitespace on commands is ignored (but not if you're within a code block).

In the following list of commands, if a command is indicated as taking an optional **[, name]** argument, this means the result of the command will go into a variable with the indicated name if the name is present; otherwise, the results are printed to stdout.

Default values of things are indicated by the following font: default value.

Assignment

An assignment is of the form **a = b** where **a** must be a valid python variable name and **b** must be an expression or a string representing a number with an optional uncertainty.

A useful feature is that you can change the coordinates of an object using an assignment and an attribute of the object. For example, suppose **pt** is a variable that is a Point object. Then you can change the x coordinate of this point by using the assignment

```
pt.x = 4
```

The Cartesian, cylindrical, and spherical coordinates are attributes of Point objects. Since Lines and Planes are defined by two Point objects (called p and q), you can change the nature of lines and planes too by changing their points. Point attributes that can be changed are `x`, `y`, `z`, `rho`, `theta`, `r`, and `phi` (see the **Notation** section for their meanings). If you change or use the angular attributes `theta` or `phi`, it's important to remember that these attributes are in the current angle unit and angle mode (i.e., compass mode, elevation mode, or negative mode corresponding to the commands **compass**, **elev**, and **neg**).

Defining objects

Points can have mass; you assign a mass by using the **m** keyword. If one or more points have a nonzero mass, then the centroid's location and total mass are printed out when the **centroid** command is used. The code allows the mass of a Point to be any object, not just a number. This allows you to perform calculations, then have the resulting points carry some information needed for further processing. Of course, the centroid and total mass won't be calculable.

In the following, when a command has a **name** parameter, a variable by that name will be created with the indicated object.

Command	Details
. x, y, [z,] name [, kw]	Define a point using rectangular coordinates. Keywords: **m** for mass. If the z coordinate is omitted, it is set to zero.
< r, theta, [z,] name [, kw]	Define a point using polar coordinates in the plane or cylindrical coordinates in space. Keywords: **m** for mass. If the z coordinate is omitted, it is set to zero.
<< r, theta, phi, name [, kw]	Define a point using spherical coordinates. Keywords: **m** for mass. **phi** is measured conventionally from the +z axis (i.e., if **phi** is zero, the point lies on the z axis).

Command	Details
| pt1, pt2, name **| pt, ln, name [, len=L]** **| pl1, pl2, name**	Define a line. You can also consider the line a vector, as it has a magnitude (its length) and direction. The first form defines a line segment extending from the two points. The second form defines a line that goes through the indicated point and has direction defined by the line **ln**. If the keyword **len** is given, then the line is of length L from the indicated point in the direction of the line **ln.** The third form is the line defined by the intersection of two planes. The line's orientation will be defined by the cross product of the normals of the two planes. Note a line can be defined by two points that are the same; this results in the "zero" line. It behaves like the zero vector and you won't be able to do things like get its direction cosines, unit vector, etc. However, it will work in cross and dot products and can result from e.g. taking the cross product of a line with itself.
ijk	This is a convenience command that calculates **in the current coordinate system** the Cartesian unit vectors (actually, lines) **i**, **j**, and **k**, the coordinate planes **xy**, **xz**, and **yz**, and the origin point **o**. If you execute a transformation, they will, in general, no longer be the unit vectors for the current coordinate system.
plane pt1, pt2, pt3, name **plane pt, ln1, ln2, name** **plane pt, ln, name** **plane pt, pl, name** **plane ln1, ln2, name**	Define a plane. For the first form, the three points must not be collinear. For the second form, the plane's normal will be in the direction of the cross product of the two lines' direction vectors and the plane will contain the indicated point. For the third form, the plane will contain the point and the line is its normal. For the fourth, the new plane will contain the indicated point and be parallel to the given plane. For the fifth, if the lines intersect, then the plane will contain the lines and the cross product defines the plane's normal. If they don't intersect, the plane contains the first line and is parallel to the second.

Transformations

These transformations change the current coordinate system in some way.

Command	Details
pop [name]	Pop a coordinate system off the stack and use this as the current coordinate system. If **name** is present, put the popped coordinate system into the variable named **name**. (What is actually popped off the stack is the 16-element coordinate transformation matrix.)
push [name]	Push the current coordinate system (i.e., coordinate transformation matrix) onto the stack. If **name** is present, use the coordinate transformation matrix referenced by the variable **name** after pushing the current coordinate transformation matrix onto an internal stack. The variable **name** must contain a coordinate system gotten with a previous **pop** command.
reset	Remove all applied transformations and use the default coordinate system. This command also deletes the contents of the internal stack.

Command	Details
rotate theta [, axis]	Rotate the coordinate system around an axis by the indicated angle. If **axis** is not given, it defaults to the z axis. If **axis** is given, it must be a line object that defines a vector direction **r**. The rotation direction is counterclockwise about the axis when the axis' unit vector points into your eye.
scale sx [, sy [, sz]]	Scale the coordinate axes measurements (i.e, a dilatation transformation). **sy** and **sz** default to 1. If only **sx** is given, the transformation is isotropic and applied to all three of the coordinate directions. For an anisotropic transformation, you must supply at least two parameters.
translate x, y [, z] **translate pt**	Translate the origin to a point. z will be zero if not given. The second form lets you use a defined point as the new origin.
xy pl **xz pl** **yz pl**	Make the argument plane **pl** parallel to the indicated coordinate plane by a rotation.

Change an object

Important: these commands will change the objects they operate on. Make copies first of any objects you wish to keep unaltered and use the following commands on the copies. The easiest way to make a copy of object A is to use an assignment command:

```
B = A.copy
```

Command	Details
length ln, L **length pl, L**	Change the length of a line or (a plane's vector) to the value L. Since a line can be considered a vector, this in reality changes the magnitude of the vector.
locate o1, o2	Take the point **o2** (or the **p**-point of a line or plane) and set the point **o1** (or the **p**-point of a line or plane) to this point. If **o1** is a line or plane, maintain the same orientation.
project pl, o1 [, o2 ...]	Project the objects onto plane **pl**. The projected objects can be points and lines. A line that is perpendicular to the plane will be projected into a point; the variable naming that line will thus subsequently hold a point. If an **eye** command has been executed with a point argument to locate the observer's eye, the projection will be a projective transformation; otherwise, it will be an orthogonal transformation.
reflect pt, o1 [, o2 ...] **reflect ln, o1 [, o2 ...]** **reflect pl, o1 [, o2 ...]**	Reflect objects about a point, line, or plane. If you want to reflect about a line, the line and all the objects must be in the xy plane.

Calculated things

Command	Details
angle pt1, pt2 [, name] **angle ln1, ln2 [, name]** **angle ln, pl [, name]** **angle pl1, pl2 [, name]**	Print the angle between two objects. For the two points, the angle is between their position vectors. If the lines don't intersect, one is parallel-transported until it intersects the other line. For two planes, the angle is the angle between the planes' normals. Between a line and a plane, the angle is the complement of the angle between the plane's normal and the line. If **name** is given, then the result is put into the

Command	Details
	indicated variable; otherwise, the result is printed. The angle is gotten from the inverse cosine of the dot product of the relevant vectors.
area pt1, pt2, pt3, [, pt4 ...]	Print the area of a polygon in the xy plane. You'll get a warning message if the polygon is self-intersecting.
centroid [pt1, pt2, ...]	If **pt1**, **pt2**, ... are given, print the centroid and mass for those points. Otherwise, print them for all points that have nonzero mass. Since you might want to use the point and mass later, they are saved in the variables `total_mass` and `centroid`.
circ3 pt1, pt2, pt3	Given three noncollinear points, print the center, radius, and diameter of the inscribed and circumscribed circles. The points must be in the xy plane.
cross o1, o2 [, name]	Calculate the cross product of two objects. If one of the objects is a point, it is interpreted as a position vector. If the object is a plane, the plane's unit normal vector will be used. All the geometric objects have vector interpretations and thus have cross products. If **name** is given, put the result into it; otherwise, print the result.
dc o1 [, o2 ...]	Prints the direction cosines and direction angles of the given objects. Direction cosines are on the closed interval [-1, 1] and direction angles are on the closed interval [0, π]. Note this is different than some textbooks, where the direction angles are on the half-open interval [0, π). For example, a position vector (0, 0, -1) will have direction angles of [π/2, π, π/2]. If you need to put the direction cosines into a variable, you can use a python command such as `dircos = object.dc` The `dc` attribute returns a 3-tuple of the direction cosines.
dist o1, o2 [, name]	Print the distance between the two indicated objects; if **name** is present, put the distance into a variable with that name. ◆ Point and point: the usual Euclidean distance. ◆ Point and line: the length of a line that is perpendicular to the given line and passes through the point. ◆ Line and line: the length of a line that is perpendicular to both lines and intersects them both. If the lines are in the same plane and not parallel, then the distance will be zero because the lines intersect. ◆ Line and plane: the line must be parallel to the plane; the distance is the perpendicular distance between the line and plane. ◆ Plane and plane: the two planes must be parallel.
dot o1, o2 [, name]	Calculate the dot product of two objects. The objects as vectors are interpreted the same way as is done for the cross product.
eye [o]	If an object is included, set the eye point for projective transformations. The object can be a point, line, or plane. If it is a line or a plane, it is the **p** point of the object that is used as the eye point. If no argument is given, it resets the eye point to None, meaning any transformations

Command	Details
	made by the **project** command are orthogonal projections.
intersect o1, o2 [, name]	Intersect the two objects **o1** and **o2**; their intersection will be one of the primitive geometric objects supported by this program. The value will be **None** if the objects don't intersect.
perp pt, ln [, name] **perp pt, pl [, name]**	Create a line perpendicular to the indicated object. For the first form, the resulting line will pass through the point and be perpendicular to the given line. For the second form, the resulting line's direction will perpendicular to the plane and point toward the point (i.e., the point will lie on a normal to the plane).
rotaxis angle, axis	Print the axis of rotation and angle of rotation in current coordinates if scale and translation transformations have not been applied into the variables named by **angle** and **axis**. If you have applied a **translation** or a **scale** command, the **rotaxis** command will fail. The printed rotation is the same as the product of all rotation transformations made up to this point.
stp o1, o2, o3 [, name]	Calculate the scalar triple product of the three objects. For three vectors ***a*, *b*, *c***, the scalar triple product is $\boldsymbol{a}\cdot(\boldsymbol{b}\times\boldsymbol{c})$.
trace pt1 [, pt2 ...]	Print the traces in current coordinates for the indicated points. The x trace is the angle between the +x axis and the point orthogonally projected into the xz plane. The y trace is the angle between the +y axis and the point projected into the yz plane. The z trace is the same as the ϕ angle in spherical coordinates. Undefined for lines or planes. The term trace as used here is analogous to the notion of the trace of a plane, which is a line of intersection of a plane with the coordinate planes [schmall:295:310]. I'll also call traces "projection angles".
vtp o1, o2, o3 [, name]	Calculate the vector triple product of the three objects. A line object will be returned. For three vectors ***a*, *b*, *c***, the vector triple product is $\boldsymbol{a}\times(\boldsymbol{b}\times\boldsymbol{c})$.

Utility or state-changing commands

In the following commands, any command listed with an argument of **[on|off]** can have an expression as an argument; if the expression is True, it is equivalent to **on**.

Command	Details
! statement	Execute an arbitrary python statement. A handy function is `out()`; it converts each of its arguments to a string and sends them to stdout. You can change the separation string, the stream the output goes to, and whether a newline is output at the end.
{ and **}**	Allow execution of an arbitrary block of python code between the braces (the braces must be on their own lines). Your code must obey python's indentation semantics. You can e.g. define variables, functions, and classes that will be accessible in subsequent code and import needed libraries.
alphabetical on|off	If you use a bare **print** command, the default behavior is to print the objects in the order they were defined. If you turn alphabetical on, the objects are sorted by name.

Command	Details
angunit const [, name]	Define the angular unit: **const** is a number that converts an angle in the desired units to radians. **name** is a string used to identify the unit in printed output. The three recognized units are: **1 rad** Radians **180/pi deg** Degrees **1/(2*pi) rev** Revolutions These units are recognized in that they will be annotated when a Point, Line, or Plane object gets printed out. However, note you can define any angular units you wish (if not the above three, you'll see an `s` in the annotation when the objects are printed, denoting a special angular unit). For example, to use gradians (i.e., the grad or gon), use `angunit 400/(2*pi), grad`.
date	Print the date and time.
del o1 [, o2 ...]	Delete named objects. It is an error if you try to delete a non-existent object.
digits n	Set number of significant digits in printed information.
eps num	Defines a number **num** such that the result of a computation whose absolute value is less than or equal to **num** will be replaced by zero. The default for **num** is 5e-15. The **eps** command exists to help minimize the aggravation of floating point calculations. For an example of the utility of **eps**, suppose you were measuring the location of points in mm and you felt that the best you could measure would be to within an uncertainty of about 0.5 mm. You might then choose **eps** to be 0.1 mm. The intent is to set small numbers to zero so they don't "pollute" subsequent calculations. If the number being compared to is a number with uncertainty, only the mean is set to zero (the uncertainty is unchanged). Warning: if you set eps too large, you may see the script stop with error messages that don't explain what's going on. You can run the script with the `-D` option to make it not catch some exceptions when dispatching a command; this can help you see where things are going wrong. Often the cause is a division by zero, which means something was set to zero because eps was large enough.
exit	Stop execution of the script.
include file	Read the commands from another file and insert them at this point. Note that the command lines aren't executed until all lines to be executed are read in.
indent num	Increase or decrease the indent level by **num**. Use 'indent 0' to set the indent level to zero (the default). This command causes subsequent printing to be indented at the current indent level. **num** can be an expression or string; the integer part of it is taken.
off **on**	**off** causes datafile lines to be ignored until an **on** command is encountered. These commands make it easy to comment out a section of the datafile.
print [o1 [,o2 ...]]	Print the indicated objects. If no arguments are given, then all objects are printed in the order that they were defined. The command line option `-a` will include variable definitions. A somewhat-screwy syntax is used for changing the output coordinate system from what is set globally: if you use `.`, `<`, or `<<` as one of the object arguments, the print command will use the associated

Command	Details
	coordinate system[5] for printing its arguments.
state	Print out the current coordinate state.

Display commands

These commands affect how objects are printed in output. The commands that have a color background also affect how subsequent input entries are interpreted. For example, if you turn compass mode on, then polar angles entered with the **<** or **<<** commands will be interpreted as compass angles.

Command	Details
2D [on|off]	If `2D` is on, points that lie in the xy plane are displayed with only two coordinates (the default is on). If you're displaying in spherical coordinates, you'll only see two coordinates for points in the xy plane, as phi is $\pi/2$; in this case, cylindrical and spherical coordinates display the same numbers. If an object has a nonzero z coordinate, all three coordinates will be shown, regardless whether `2D` is on or off. Note: you can also supply an argument that is an expression; if the expression is True, then `2D` mode is turned on.
compass [on|off]	Turns compass mode on and off; off is the default. With compass mode on, the angle θ of polar, cylindrical, and spherical coordinates is measured clockwise from the +y axis, which corresponds to the north direction. East is at 90°. This simulates the usual compass headings read from a compass.
deg	Set the angle measurement units to degrees. Degree mode is the default, but you can change this in the function `ParseCommandLine()`. You can change the default angle mode to radians by the `-r` command line argument.
elev [on|off]	Turns elevation mode on and off; off is the default. Elevation on means that the spherical coordinate ϕ is replaced by its complement angle ψ. ψ is the elevation above or below the xy plane. Elevation is also called altitude in astronomy.
neg [on|off]	Turns negative mode on and off; off is the default. Negative mode on means the angle θ of polar, cylindrical, and spherical coordinates is measured opposite to the customary direction (clockwise for regular mode and counterclockwise for compass mode, both when looking down at the origin from the +z axis). The ϕ angle of spherical coordinates is not affected by this setting.
polar **cyl**	Show output results in cylindrical coordinates. If the data are two-dimensional, the output is in polar coordinates.
rad	Use radians to measure angles.
rect	Show output results in rectangular coordinates; this is the default.
rev	Use revolutions to measure angles.
sph	Show output results in spherical coordinates. If the data are two-dimensional, the output is in polar coordinates.

5 This was done to keep the existing code relatively simple.

Geometric primitives

These geometric primitives are defined in the file `geom_prim.py` module. This module provides implementations of primitive three-dimensional geometric objects: points, lines, planes, vectors, and coordinate transformations. It is used as the basis of the `xyz.py` script.

Some features of `geom_prim.py` are:

- If the python uncertainties library is installed, the coordinates and transformations can involve uncertainties that will be propagated into the resulting calculations.
- The coordinates of points can be gotten in Cartesian, cylindrical, and spherical coordinates. The compass, neg, and elev settings are honored (class variables of Ctm).
- Intersections of these objects (when they result in a supported primitive object) can be gotten.
- If you only apply a sequence of rotation transformations, the resultant product transformation can be given by the angle of rotation and the fixed axis (eigenvector of the product rotation matrix corresponding to the eigenvalue of 1) can be calculated.
- A class variable `Ctm.eps` is defined that can be used to help with floating point comparisons of numbers that are near zero. If a calculation results in an absolute value difference that is less than `Ctm.eps`, then the result is set to zero. This is intended to deal with small roundoff errors that are inevitable with floating point calculations.

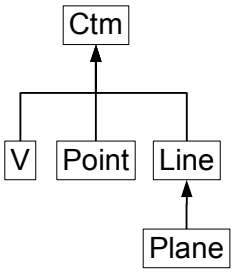
The `Ctm` base class contains the coordinate transformation matrix (CTM) which is used to convert between the current coordinate system (CCS) and default coordinate system (DCS) coordinates. Because `Ctm._CTM` is a class variable, it is common to all geometrical objects.

Something to remember about the transformations is:

> **The coordinate transformation matrix transforms from the default coordinate system (DCS) to the current coordinate system (CCS).**

The `Ctm` object has two methods to get the coordinates of an object: `ToDCS()` and `ToCCS()`. These are abstract methods of the base class and must be defined in the derived classes (because the Ctm object has no notion of coordinates, only transformation matrices). The `ToCCS()` method uses the CTM and the `ToDCS()` method uses the inverse of the CTM.

The object inheritance is as follows:



The V class is a helper class for vectors used internally. If I had things to do over again, I'd probably put all of `V`'s functionality in the Line object.

Points

Points are defined by three Cartesian coordinates; missing coordinates in the definition are zero.

Points can also be defined by the intersection of two lines and three planes.

Point equality occurs when the points have the same coordinates.

Negating a point multiplies the components by -1 (i.e, reflects it about the origin).

Attributes of points (the returned values are in the current coordinate system):

`copy`	Returns a copy of the Point object.
`proj_ang`	Returns a 2-tuple of the projection angles (traces) of the point.
`dc`	Returns a 3-tuple of the direction cosines of the point.
`rec, rect, xyz`	Returns a 3-tuple of the Cartesian coordinates of the point.
`cyl`	Returns a 3-tuple of the cylindrical coordinates of the point.
`sph`	Returns a 3-tuple of the spherical coordinates of the point.
`x`	Returns the x coordinate. Settable.
`y`	Returns the y coordinate. Settable.
`z`	Returns the z coordinate. Settable.
`rho`	Returns the rho coordinate of cylindrical coordinates. Settable.
`r`	Returns the r spherical coordinate. Settable.
`theta`	Returns the azimuthal angle in polar, cylindrical, and spherical coordinates. Settable.
`phi`	Returns the phi coordinate of spherical coordinates (angle from +z). Settable.

The angular attributes are returned in the current angular unit and honor the `_compass, _elev,` and `_neg` class variables of `Ctm`.

Lines

Lines are defined by two points. Suppose the two points have position vectors $\boldsymbol{p}$ and $\boldsymbol{q}$. Then a unit vector in the line's direction is

$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{q}-\boldsymbol{p}}{|\boldsymbol{q}-\boldsymbol{p}|}$$

A parametric equation of the line with parameter t is

$$\boldsymbol{r} = \boldsymbol{p} + t\,\hat{\boldsymbol{u}}$$

Note that the unit vector's components are the direction cosines of the line. Internally, a line is initialized with the beginning point **p** and the ending point **q**.

When using a line for an axis of rotation, the rotation direction is counterclockwise when the tip of the line's unit vector hits you in the eye (i.e., the vector is pointing at you).

A line can also be defined by the intersection of two planes. The direction of the line is defined by the cross product of the planes' normals.

Negating a line reverses the direction of the line.

Lines behave as vectors and can be added and subtracted. They can be multiplied and divided by scalars. They also have dot and cross products.

Line equality occurs when the two lines have the same points p and q.

The normalize() method changes the line's length to unity.

Attributes of lines:

`copy`	Returns a copy of the Line object.
`dc`	Returns a 3-tuple of the direction cosines of the line.
`L`	Returns the length of the line. You can get the same thing by taking the absolute value of the line.
`p`	The beginning point of the line. Settable.

`q` The ending point of the line. Settable.

`u` Unit vector of the line's direction.

Planes

Planes can be defined by

- Three noncollinear points
- A point and two lines. The plane passes through the point and is parallel to the plane defined by the two lines. The plane's normal is parallel to the cross product of the unit vectors of the two lines.
- A point and a line (the plane contains the given point and line)
- A point and a plane (the plane contains the point and is parallel to the given plane)
- Two lines that intersect. The normal to the plane is determined by the cross product of the line's vectors.
- A line. The p attribute of the line is in the plane and the line is in the direction of the plane's normal.

Negating a plane multiplies the normal vector by -1.

A Plane object inherits from a Line object, so all methods and attributes available to Lines are available to Planes. Additional attributes are:

`dnd` Returns the directed normal distance from the origin to the plane. This is usually called p in the Hessian normal form equation for a plane.

`n` Returns a normal vector to the plane. Note this is a V object (a vector in `geom_prim.py`). Contrast to the `u` attribute which returns a 3-tuple.

Math

It is not necessary to understand the math behind the `xyz.py` script to be able to use its features. Virtually all of the math is at the level of analytical geometry taught in a college freshman calculus class. A few topics (rotation matrices and homogeneous coordinates) might not be in a calculus text, but are understandable with the many web pages out there explaining them.

Two-dimensional case

A two-dimensional coordinate system using Cartesian coordinates (x, y) for points has the following affine transformations (*a*, *b*, *c*, *d*, *t*, and *u* are constants):

$$x' = ax + by + t$$
$$y' = cx + dy + u$$

It is convenient to represent these transformations by a square matrix:

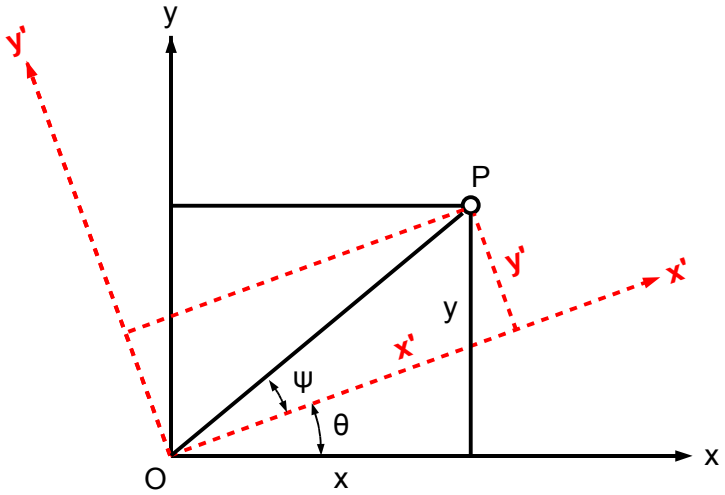
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t \\ c & d & u \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where the third "coordinate" 1 introduces the notion of "homogeneous coordinates"; this is an artifice to allow matrix notation to be used with translations, which are nonlinear transformations. Thus, the effect of two transformation matrices is found by (left) multiplying them together with matrix multiplication. Since matrix multiplication is non-commutative in general, the resulting transformation depends on the order of the component transformations. Expanding the determinant of the matrix along the third row, we see the determinant of the transformation is $ad - bc$, which makes the transformation invertible if the determinant is nonzero.

The `rotate` command results in the transformation matrix

$$\mathbb{R}_z = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where θ is the rotation angle. It's useful to see the derivation of this (passive) transformation:



Consider the point P fixed in space and we rotate the coordinate system about the origin O to the new x'y' system. Let r be the distance OP. We have

$$x = r\cos(\psi + \theta) = r(\cos\psi\cos\theta - \sin\psi\sin\theta)$$

Since $\sin\psi = y'/r$ and $\cos\psi = x'/r$, we get

$$x = x'\cos\theta - y'\sin\theta \tag{1}$$

Similarly

$$y = r\sin(\psi+\theta) = r(\sin\psi\cos\theta + \cos\psi\sin\theta)$$

which leads to

$$y = x'\sin\theta + y'\cos\theta \tag{2}$$

Solving (1) and (2) for x' and y', we get

$$\begin{aligned} x' &= x\cos\theta + y\sin\theta \\ y' &= -x\cos\theta + y\cos\theta \end{aligned}$$

which is the same as the transformation matrix given above (ignoring the third row and column).

The rotation matrix as defined transforms a set of coordinates (x, y) to the coordinates in the new coordinate system (x', y'). If you want to go from (x', y') to (x, y), use the inverse of the matrix $\mathbb{R}_z$, which is a rotation matrix. Since rotation matrices are orthogonal, you just take the transpose to get the inverse[6] (a very handy property, as you don't have to calculate the inverse using the usual calculational methods). These matrices are representations of the special orthogonal group SO(2), assuming you delete the third row and column in $\mathbb{R}_z$.

The `translate t, u` command results in the transformation matrix

$$\begin{bmatrix} 1 & 0 & -t \\ 0 & 1 & -u \\ 0 & 0 & 1 \end{bmatrix}$$

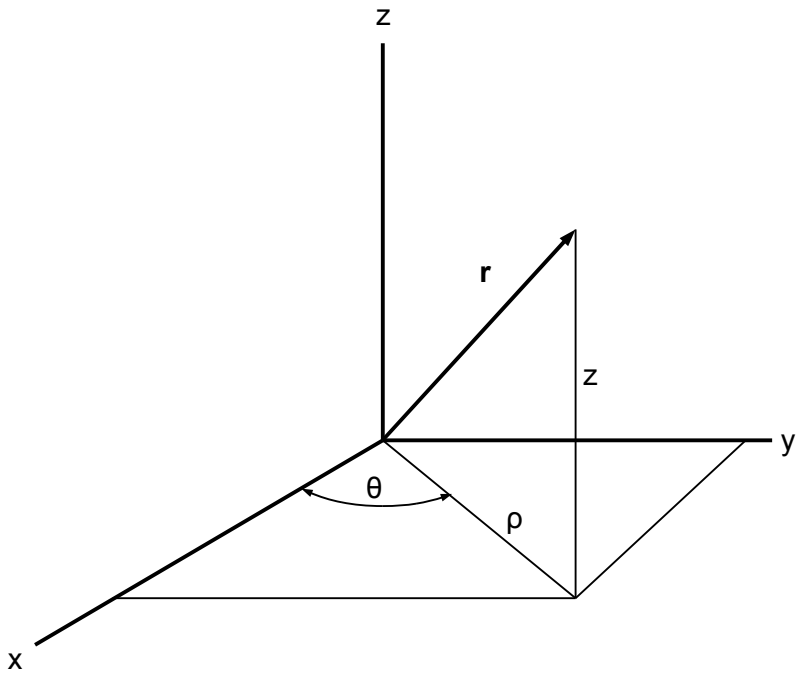
The `scale` s_x, s_y command results in the transformation matrix

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

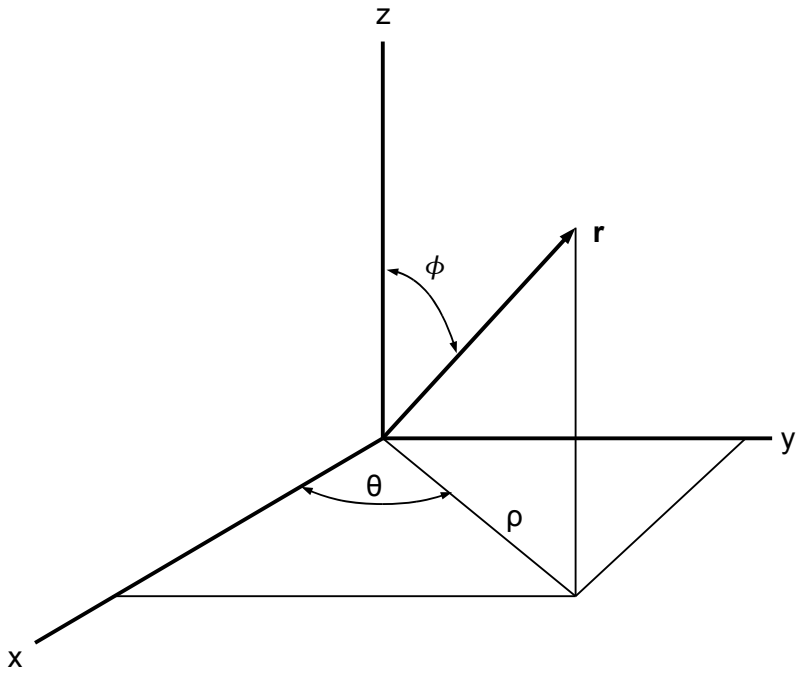
Three-dimensional case

The coordinates we use are the usual right-handed Cartesian, cylindrical, and spherical coordinate systems. For cylindrical coordinates, we have the cylindrical coordinates (ρ, θ, z) that locate the point ***r***:

6 Another way of inverting rotation matrices is to substitute -θ for θ.



For spherical coordinates [sph], we have (r, θ, ϕ):



The domains of the arguments used in the `xyz.py` script are (in radians):

$$r \in (0, \infty)$$
$$\theta \in [0, 2\pi)$$
$$\phi \in [0, \pi/2]$$

Note these are slightly different than the usual mathematical definitions. If ***r*** is zero, then the point is set to the origin; the cylindrical coordinates of the origin in cylindrical coordinates are (0, 0, 0) and in spherical coordinates and radians (0, 0, π/2).

The relevant affine transformations are

$$x' = ax + by + cz + t$$
$$y' = dx + ey + fz + u$$
$$z' = gx + hy + iz + v$$

where *a* through *i*, *t*, *u*, and *v* are constants. The transformation in homogeneous coordinates is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & t \\ d & e & f & u \\ g & h & i & v \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The (augmented) rotation matrices for rotations around each of the axes are

$$\mathbb{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbb{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbb{R}_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A point P with coordinates $(x, y, z, 1)$ in the original coordinate system will have the new coordinates in the rotated coordinate system gotten by multiplying the relevant matrix by the column vector $(x, y, z, 1)^T$. A positive angle θ is in the counterclockwise direction when you look towards the origin down the positive part of the axis (i.e., when the axis' unit vector's tip hits you in the eye). These rotation matrices are representations of the special orthogonal group in 3 dimensions SO(3). *Special* means the determinant of the matrix is +1 and orthogonal means the matrix inverse is equal to the transpose (delete the fourth column and row to get the SO(3) representations).

The translation transformation matrix is

$$\begin{bmatrix} 1 & 0 & 0 & t \\ 0 & 1 & 0 & u \\ 0 & 0 & 1 & v \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the dilatation (scaling) transformation matrix is

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note these matrices, in general, don't commute. For example, place a book in front of you, rotate it through a right angle about the yaw axis and then rotate it again through a right angle about the pitch axis. If you do these rotations in reverse order, the final orientation of the book won't be the same as the first set of movements. Euler's theorem states that an arbitrary number of different rotations are equivalent to one rotation around one axis; the `xyz.py` script can find the angle and axis of such a set of rotations by the **rotaxis** command.

If you've been exposed to infinitesimal Lie transformation groups, the easiest way to see that these transformations don't commute is to look at their infinitesimal generators: ∂_x, $x\partial_x$, and $x\partial_y - y\partial_x$. You can see by inspection that they don't commute.

Polygon self-intersection

This section gives a description of the algorithm used to determine if a polygon given as an argument to the **area** command intersects itself. We'll assume all of the points are in the xy plane.

Suppose the vertices are given as an ordered set $(p_1, p_2, \ldots, p_n)$ where $p_1 = (x_1, y_1)$, etc. Note that the **area** command only works for points in the xy plane (thus, if you want to use it with some points

in space, you can use the **project** command to first project the points into the xy plane or translate and rotate things to get them as needed. The algorithm is to construct the array

$$\begin{bmatrix} p_1 & p_2 \\ p_2 & p_3 \\ \dots & \dots \\ p_n & p_1 \end{bmatrix}$$

in which each row represents the two endpoints in a side of the polygon. Then all combinations of the rows of this array taken two at a time are examined for their intersection point. If the intersection point lies between both of the pairs of endpoints defining a side (but not including them, meaning the parameters are both in the open interval (0, 1)), then self-intersection has occurred.

The parametric equations for a line passing through two points (x_1, y_1) and (x_2, y_2) in the Cartesian plane are

$$\begin{aligned} x &= x_1 + (x_2 - x_1)t \\ y &= y_1 + (y_2 - y_1)t \end{aligned}$$

These equations yield (x_1, y_1) for $t = 0$ and (x_2, y_2) for $t = 1$.

To find the intersection point of two lines given by the parametric equations

$$\begin{aligned} x &= x_1 + (x_2 - x_1)t \\ y &= y_1 + (y_2 - y_1)t \end{aligned} \quad \text{and} \quad \begin{aligned} X &= X_1 + (X_2 - X_1)s \\ Y &= Y_1 + (Y_2 - Y_1)s \end{aligned}$$

we equate $x = X$ and $y = Y$ and solve for the values of the parameters t and s:

$$\begin{aligned} D \cdot t &= X_2(y_1 - Y_1) + x_1(Y_1 - Y_2) + X_1(Y_2 - y_1) \\ D \cdot s &= x_2(y_1 - Y_1) + x_1(Y_1 - y_2) + X_1(y_2 - y_1) \end{aligned}$$

where

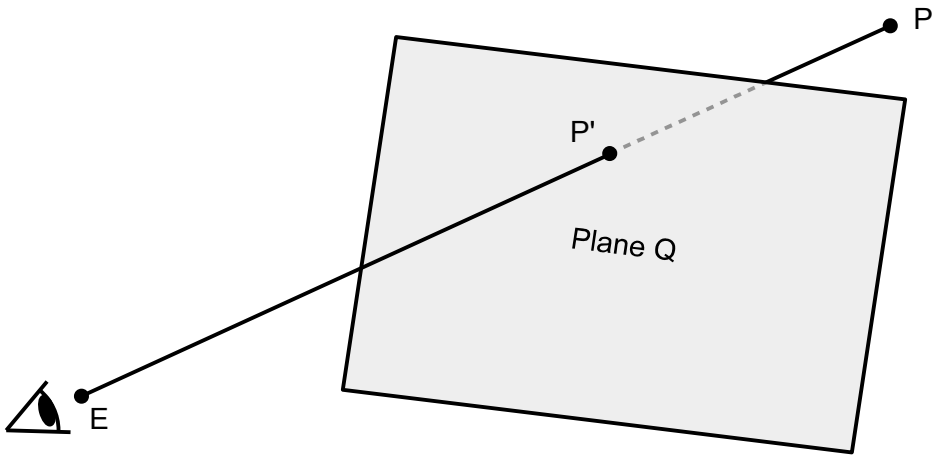
$$D = \begin{vmatrix} x_1 - x_2 & X_1 - X_2 \\ y_1 - y_2 & Y_1 - Y_2 \end{vmatrix}$$

A check with the two lines through the points [(0, 0) , (1, 1)] and [(0, 1), (1, 0)] gives 0.5 for both parameters for the intersection point, as expected. Since both of these parameters are in the open interval (0, 1), the lines intersect and a polygon with these lines for sides would be declared to be self-intersecting.

Projections

The `xyz.py` script supports two types of projection: orthogonal and projective transformations. Orthogonal projection means a line or point is projected onto a plane by putting the eye off at infinity so that the projecting rays arrive normal to the plane. This type of transformation is commonly used in engineering drawings.

Projective transformations require a finite eye point E. The point P to be projected into a plane Q then causes a line between the points E and P to be constructed; the intersection of this line and the plane Q results in a point P', which is the projected point. Such projective transformations have the characteristic that points further from the plane Q (and on the plane's side opposite to the eye point) are closer together when projected into the plane. These projective transformations work for any points in space except those on a line EP that is parallel to the plane Q (which puts the projected point off at infinity).



Note point P can be on either side of the plane; there are no restrictions or viewports as is common with projections for computer graphics. If you move the eye point E off infinitely far from the plane, you'll have orthogonal projections.

References

I don't give any references to a basic calculus text because there are so many of them available and it's unlikely you'll have access to the books I have. Probably any popular text should be suitable, assuming the basic calculus topics haven't changed since I was a student.

If you have access to Google books, there are a number of books on analytic geometry and vectors from the early 1900's. These have underlined reference abbreviations.

an `AnalyticGeometry.pdf` at http://code.google.com/p/hobbyutil/, a document that summarizes a number of formulas from analytical geometry.

coff J. Coffin, *Vector Analysis*, Wiley, 2nd ed., 1911. Suitable for self-study if you've had beginning college calculus and physics courses.

crc W. Beyer, *CRC Standard Mathematical Tables*, 26th ed., CRC Press, 1981. Page 355 has some useful vector formulas for the 3D geometry of planes and lines.

ga C. Doran and A. Lasenby, *Geometric Algebra for Physicists*, Cambridge University Press, 2003.

gum http://www.bipm.org/en/publications/guides/gum.html *Evaluation of Measurement Data -- Guide to the Expression of Uncertainty in Measurement*, Working Group 1 of the Joint Committee for Guides in Metrology, 2008 (1995 version with minor corrections). This has become the standard document for how to express uncertainty in measurement; it has played an important role in filling the need to standardized the reporting and interpretation of uncertainty in physical measurements.

osgood W. Osgood and W. Graustein, *Plane and Solid Analytic Geometry*, Macmillan, 1921.

ps Adobe, *PostScript Language Reference Manual*, 2nd ed., Addison-Wesley, 1990 (usually called the red book). As PostScript deals with a two-dimensional space (a sheet of paper), it uses a coordinate transformation matrix of 6 elements. But the basic ideas are the same as can be used in three-dimensional space, so reviewing the coordinate transformation section(s) could prove useful.

punc https://pypi.python.org/pypi/uncertainties/ This python library allows calculations with numbers that have uncertainties and does linear uncertainty propagation.

rm The topic of rotation matrices is not typically discussed in the freshman math class, but the ideas are accessible if you're willing to read a bit. To understand the `xyz.py`'s code, all you'll really need is to understand the generation of the rotation matrix given the desired angle to rotate and the axis to rotate about (the formula is here: http://en.wikipedia.org/wiki/Rotation_matrix). Also see [an].

Some links:

http://en.wikipedia.org/wiki/Euler%27s_rotation_theorem A theorem that states that a sequence of rotations is equivalent to one rotation about a particular axis.

http://en.wikipedia.org/wiki/Charts_on_SO%283%29 Discusses different coordinate systems when discussing rotations of three-dimensional space.

http://en.wikipedia.org/wiki/Rotation_formalisms_in_three_dimensions Looks at different ways of parameterizing rotations. Unfortunately, there are numerous conventions and notations.

schmall C. Schmall, *A First Course in Analytic Geometry*, 2nd ed., van Nostrand, 1921.

snyder V. Snyder and C. Sisam, *Analytic Geometry of Space*, Holt, 1914.

sph http://en.wikipedia.org/wiki/Spherical_coordinates Note there are different conventions for spherical coordinates, so it's important you know which convention is being used. I

use the mathematical definition where θ is the polar azimuth angle in the xy plane, the same as in cylindrical coordinates, and the angle ϕ is the angle from the +z axis.

wilson E. Wilson, *Vector Analysis*, Yale Univ. Press, 1922. This is based on Gibbs' lectures from the 1880's.